VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Specializovaný framework nad ASP.NET

Specialized Framework for ASP.NET

2010 Jakub Macek

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava*.

Tato diplomová práce má následující neveřejně části, umístěné na samostatném CD:

1. zdrojové kódy vytvořeného frameworku
2. zdrojové kódy a designové podklady ukázkových případů

V Ostravě 7. května 2010 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2010 .....

Rád bych na tomto místě poděkoval svému vedoucímu Ing. Tomáši Frydrychovi, zejména za jeho pomoc s formálními stránkami vypracování, a Ing. Tomáši Poskerovi za podporu tohoto projektu. Také bych chtěl poděkovat kolegům z firmy Poski.com za jejich zpětnou vazbu při vývoji frameworku.

Abstrakt

Tato práce se zabývá metodami výroby webových aplikací a speciálně webových prezentací v prostředí ASP.NET. Jejím cílem je najít a analyzovat klíčové problémy spojené s touto činností a následně navrhnout a implementovat řešení v podobě specializovaného frameworku pro ASP.NET. V prvních kapitolách popisuje novou architekturu pro webové aplikace. V navazujících kapitolách řeší problémy lokalizace, vstupu uživatelských dat a jejich persistence v databázi nebo ve vyrovnávací paměti. Práce je uzavřena řešeními tří případů z praxe.

Klíčová slova: webová aplikace, webová prezentace, ASP.NET, framework, .NET, C#, lokalizace, persistence, formuláře, cache

Abstract

This thesis deals with methods of production of web applications and focuses especially on ASP.NET based web presentations. Its goal is to discover and analyze problems related to this pursuit and subsequently design and implement solution to these problems in a form of specialized framework for ASP.NET. Opening chapters describe new architecture for web applications. Following chapters solve problems of localization, user input and data persistence both in database and cache. Thesis is concluded with solutions to three practical cases.

Keywords: web application, web presentation, ASP.NET, framework, .NET, C#, localization, persistence, forms, cache

Seznam použitých zkratek a symbolů

ASP.NET	–	Advanced Server Pages .NET, prostředí pro vývoj webových aplikací v .NET
ASP.NET MVC	–	MVC framework od firmy Microsoft pro ASP.NET
CMS	–	Content Management System, informační systém pro správu obsahu
CRM	–	Customer Relationship Management, informační systém pro řízení vztahu se zákazníky
CRUD	–	Základní operace s daty: Create, Read, Update, Delete
HTML	–	HyperText Markup Language, značkovací jazyk webových stránek
HTTP	–	HyperText Transfer Protocol, protokol používaný pro přenos webových stránek
JSON	–	JavaScript Object Notation, forma zápisu a serializace dat
LINQ	–	Language Integrated Query, dotazovací jazyk integrovaný v .NET
MVC	–	Model-View-Controller, aplikační architektura
MSDN	–	Microsoft Developer Network
PHP	–	Skriptovací jazyk pro tvorbu webových aplikací
SEO	–	Search Engine Optimization
SQL	–	Dotazovací jazyk používaný v relačních databázích
URL	–	Uniform Resource Locator, označení zdroje v rámci sítě Internet
WYSIWYG	–	What You See Is What You Get

Obsah

Seznam obrázků

1 Úvod

Pro webové prezentace a aplikace menšího rozsahu bývá, zejména na českém trhu, typicky implementačním jazykem PHP a prostředím webový server Apache běžící nad některou z distribucí operačního systému Linux. Tento jazyk umožňuje snadné vybudování základů webové aplikace a ze své podstaty také její případné násilné přiohnutí, pokud je třeba provést radikální změnu.

Díky tomu pro tuto oblast vznikla řada frameworků, pomocí kterých se vývoj značně urychlí. Naneštěstí tytéž vlastnosti, které umožňují programátorovi jazyk a prostředí rychle pochopit na základní úrovni, jako například slabé typování a nedořešený objektový model, mu s přibývajícími zkušenostmi a nároky na výslednou aplikaci výrazně znesnadňují její vývoj a údržbu. V řadě případů tak vznikají řešení, která sice splňují definované požadavky, ale lze je jen těžko označit za čistá.

Absence nástrojů jako jsou refactoring nebo pokročilá práce s databází a business entitami nutí vývojáře přecházet do pokročilejších prostředí založených na jazyku Java nebo prostředí ASP.NET, kde je kvalitní podpora pro vývoj enterprise aplikací a informačních systémů. Z historických důvodů a také díky své otevřenosti je v prostředí jazyka Java širší nabídka hotových nebo předpřipravených řešení a tedy i větší šance, že programátor nalezne takové, které odpovídá potřebám jeho projektu. Mladší prostředí ASP.NET i přes svůj rychlý a dynamický rozvoj neposkytuje v současné době dostatečné prostředky pro tvorbu webových prezentací a webových aplikací se důrazem na prezentační část.

Díky přirozené uzavřenosti platformy ASP.NET, resp. celého frameworku .NET, jsou tyto překážky hůře překonatelné. Příkladem může být URL routování (metoda snadného dosažení konzistentních a dobře optimalizovaných URL pro stránky webu), které nahradilo dřívější méně dokonalou techniku jednosměrného přepisování URL pomocí mod_rewrite (modul HTTP serveru Apache určené k tomuto účelu).

V PHP komunitě došlo k rychlé adopci tohoto konceptu a dnes si můžeme vybrat z desítek open-source frameworků, které jej podporují[1]. Na druhou stranu v prostředí ASP.NET již takový výběr není a existující zdarma dostupné frameworky jako Monorail vzniklý v roce 2007 nebo v roce 2009 uvolněný ASP.NET MVC od Microsoftu.

1.1 Cíle práce

Při návrhu a implementaci tohoto frameworku jsem vycházel především z potřeb při vytváření webových prezentací, které jsem zaznamenal během své dosavadní praxe s vytvářením komerčních projektů v PHP. Poptávka na trhu v této oblasti vyžaduje od výrobců snížení nákladů a vysokou flexibilitu ve řízení změn požadavků. Tento framework se tedy především specializuje na zkrácení času potřebného k výrobě první funkční verze prezentace a zároveň se snaží zvýšit podíl neprogramátorů na projektu a umožnit programátorovi podílet se na větším množství projektů současně.

[1] z nich byl uvedl například framework Symfony začínající v roce 2005 (http://en.wikipedia.org/wiki/Symfony_framework) a PRADO (http://www.pradosoft.com/about/) vzniklý v roce 2004

Tato práce si klade za cíl vytvořit základní rámec pro prostředí ASP.NET, který bude schopen i vývojářům teprve začínajícím s tímto prostředím, poskytnout specializované řešení na klíčové problémy pro tvorbu webových prezentací v následujících oblastech:

1. Vytváření statických částí webové prezentace (tedy takových, co nepodléhají správě obsahu).
2. Optimalizace pro vyhledávací stroje s individuálními URL pro jednotlivé jazyky.
3. Persistence dat.
4. Vstup dat pomocí formulářů.

První část práce poskytne čtenáři náhled to teoretických konceptů, na jejichž základě fungují webová aplikace obecně a v prostředí ASP.NET. Bude zde také popsána podstata problémů, se kterými se lze setkat, a navrženo několik vhodných řešení.

V druhé části budou ke každému tématu objasněn důvod, proč je třeba se jím při vývoji zaobírat, a jaké výhody plynou z úspěšného vyřešení souvisejících problémů. Dále bude provedeno srovnání dostupných řešení, ať už teoretických, či praktických, následně návrh a popis implementace vlastního řešení s několika krátkými, praktickými příklady.

Součástí práce jsou tři řešené případy reálných webových prezentací, na kterých se může čtenář rychle a snadno seznámit s nejdůležitějšími a nejužitečnějšími vlastnostmi frameworku.

1.2 Použité termíny

Pro mnoho anglických termínů z oblasti webový aplikací neexistují adekvátní české termíny, nebo nebyly dosud přijaty českou programátorskou veřejností. Proto budou v takových případech použity v zájmu jednoznačnosti a srozumitelnosti počeštěné varianty termínů anglických.

2 Teoretická východiska

2.1 Webová prezentace

Webová aplikace je takový software, který běží centralizovaně na serveru a umožňuje přístup uživatelům skrze počitačovou síť. Tato práce se zabývá takovými webovými aplikacemi, které jsou poskytovány skrze celosvětovou síť Internet a jejich cílem je především prezentovat svého provozovatele a jeho činnost, služby, produkty či reference. Takové webové aplikace se označují jako webové prezentace a jejich aplikační logika se z větší části skládá ze zobrazování a správy obsahu.

Z tohoto cíle vycházejí důležité vlastnosti webových prezentací, přede všemi ostatními dostupnost informací co největšímu množství uživatelů. Důsledkem toho je třeba prezentaci co nejlépe ošetřit nejen ze strany pro návštěvníka, ale také uživatelské rozhraní pro správu obsahu musí být připraveno jednoduše a srozumitelně, neboť správci obsahu bývají v mnoha případech uživatele-začátečníci.

2.2 ASP.NET

ASP.NET je součást frameworku Microsoft .NET, která umožnuje vývoj webových aplikací v tomto prostředí. Obsahuje velké množství tříd, které řeší nejdůležitější činnosti související s provozem webové aplikace, od zpracování příchozího HTTP požadavku až pod výstup k uživateli.

2.2.1 WebForms

Integrovanou součástí prostředí ASP.NET jsou WebForms [1]. Jde o základní architekturu webové aplikace, kde v adrese HTTP požadavku je uvedena přímo cesta k souboru, který se má vykonat. Toto řešení je běžné i v dalších prostředích, například PHP, Perl/CGI nebo JSP v jazyku Java.

WebForms vycházejí koncepčně z WinForms. Každá stránka je analogii okna (anglicky označované jako form) v desktopové aplikaci. Další společnou vlastností je, že programování WebForms je řízeno událostmi.

Implementace stránky se tedy skládá ze dvou částí - návrhu a implementace událostí. Návrh může být tvořen pomocí speciálního nástroje podobně jako u oken desktopových aplikací, nebo může být zapisován v HTML s rozšířenou sadou značek. Implementace událostí je vytvořena v jazyce C# nebo VisualBasic.NET.

2.2.2 ASP.NET MVC

Pokročilejším řešením je ASP.NET MVC (http://www.asp.net/mvc/). Jde o samostaný framework založený na třívrstvém architekturálním vzoru Model-View-Controller.

Model představuje samotnou aplikační logiku, která je nezávislá na způsobu, jakým s ní bude pracovat uživatel. Práce s databází nebo výpočty nad daty jsou typickými příklady činností, které se provádí v modelu. V architektuře je implementace modelu výhradně záležitostí programátora a MVC framework do ní nazasahuje. Pro model je

tedy možné si vybrat libovolnou jinou podpůrnou knihovnu (například LINQ to SQL nebo ADO.NET)

Naopak implementace vrstvy Controller je vyžadována ve striktní formě, protože tato vrstva má za úkol přebírat požadavky od uživatele a přípravit pro něj odpovědi. Ke zpracování může ale nemusí používat vrstvu Model a výsledek své činnosti postoupí vrstvě View.

Vrstva View je také zpracovávana frameworkem a proto musí dodržovat určitá pravidla. Na svém vstupu přebírá data od vrstvy Controller a jejím úkolem je vytvořit odpověď, které bude rozumět webový prohlížeč uživatele (tzn. HTML stránku, soubor ke stažení apod.).

Tato práce používá některé koncepty MVC, jako vrstvu View, ale kvůli své specializaci například stanovuje striktnější omezení například v modelu.

2.3 Teorie v kapitolách

Pro větší návaznost a srozumitelnost textu jsou teoretická východiska rozdělena přímo k tématům, kterými se práce zabývá. Jednotlivé kapitoly začínají samostatnými sekcemi, které obsahují teoretické i praktické základy, na kterých jsou dnes webové aplikace (a speciální webové prezentace) založeny a rozbor nejvíce rozšířených existujících řešení.

3 Architektura a životní cyklus požadavku

3.1 Životní cyklus požadavku v ASP.NET

Při vstupu HTTP požadavku do ASP.NET aplikace je z něj vytvořena instance třídy HttpRequest. Ta je společně s prázdnou instancí HttpResponse a dalšími daty zabalena do nové instance třídy HttpContext. K HttpContext je připojena instance HttpApplication (nebo třídy, která je z ní odvozena), které následně předáno řízení. Každá webová aplikace v ASP.NET obsahuje nejvýše jednu třídu odvozenou od HttpApplication a její implementace musí být v souboru Global.asax v kořenovém adresáři aplikace. Tato třída ale může mít mnoho instancí, přičemž každá instance může být následně znovupoužita pro zpracování více požadavků.

Instance HttpApplication má za úkol spouštění událostí v rámci životního cyklu a spouštění modulů a handlerů definovaných v souboru Web.config. Zatímco požadavek může projít několika moduly, tak handler je každému požadavku přiřazen pouze jeden a stojí na konci řetězu zpracování. Třída implementující rozhraní IHttpHandler [2] obdrží ke zpracování instanci HttpContext obsahující požadavek.

Framework se do prostředí ASP.NET integruje na dvou místech:

1. Implementací vlastního handleru, který je nutno přiřadit v rámci Web.config
2. Vlastní rodičovskou třídou, od které musí programátor odvodit svoji třídu aplikace v Global.asax.

3.1.1 HTTP handler

Pro zpracování požadavku frameworkem je třeba nasměrovat všechny nezpracované požadavky na třídu InvocationHttpHandler. Vložení handleru do seznamu lze provést upravením souboru Web.config tak, aby obsahoval následující definici. Důležité je přidání řádku s elementem <add> do sekce s handlery. Podle použitého webového serveru a jeho nastavení to může být v sekci system.web/httpHandlers nebo system.webServer/handlers. Pokud existují obě, je vhodné přidat handler frameworku do obou.

```
<system.web>
  <compilation debug="true"/>
  <httpHandlers>
    <add verb="*" path="*" validate="false"  type="PoskiNET.InvocationHttpHandler, PoskiNET"/>
  </httpHandlers>
</system.web>
```

Tento handler provádí tři základní činnosti:

1. Inicializaci aplikace.
2. Posílání souborů, které není třeba zpracovávat (jde o soubory v adresářích _css/, _js/, _images/).
3. Vytvoření a spuštění instance InvocationContext.

3.1.2 InvocationHttpApplication a InvocationApplication

Inicializace aplikace je prováděna voláním statické metody Initialize() ve třídě InvocationApplication. Ta obsahuje data, která budou sdílena při zpracování všech požadavků.

Inicializace je rozdělena na několik fází, přičemž konfigurace různých částí aplikace je popsána dále v příslušných kapitolách. Základní nastavení jsou vždy provedena v rámci frameworku a ten následně vyvolá příslušnou instanční metodu třídy aplikace z Global.asax. Jsou vyvolávány v následujícím pořadí.

1. InitializeApplicationLocales() - Definice jazyků aplikace, viz. kapitola o lokalizaci.
2. InitializeApplicationEntities() - Definice entitních typů, viz. kapitola o persistenci.
3. InitializeApplicationModules() - Definice modulů, viz. sekce v této kapitole.
4. InitializeApplicationTranslations() - Nahrání doplňkových lokalizačních souborů, viz. kapitola o lokalizaci.
5. InitializeApplicationInvocations() - Definice a nastavení vyvolání, viz. sekce v této kapitole.
6. InitializeApplicationRouter() - Definice routování, viz. sekce v této kapitole.
7. InitializeApplicationView() - Definice nových poskytovatelů vykreslení pro šablony, viz. kapitola o vrstvě View.
8. InitializeApplicationFinished() - Po dokončení všech inicializací.

3.2 Kontext vyvolání

Podobně jako instance třídy HttpContext poskytuje přístup k informacím týkajícím se HTTP požadavku, tak instance InvocationContext v sobě shrnuje dodatečné informace ke zpracování vyvolání a zároveň obaluje instanci HttpContext.

Obdobně jako v HttpContext jsou všechny vlastnosti připraveny frameworkem a programátor aplikace pouze využívá jejich hodnoty.

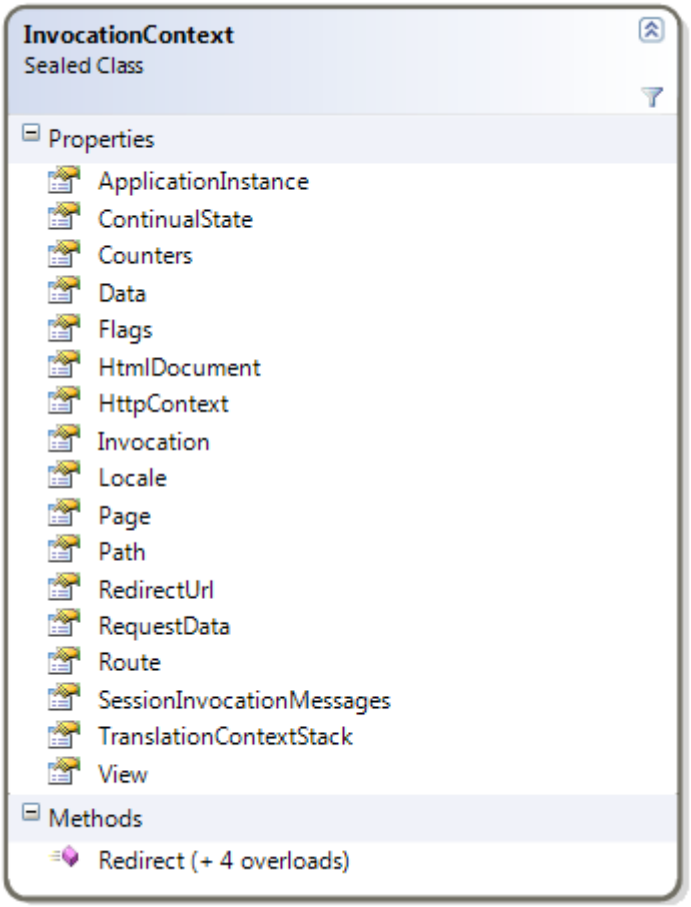
Jedinou metodou InvocationContext, kterou může programátor využít je metoda Redirect(), která slouží k nastavení přesměrování. Přesměrování se provádí až v rámci fáze výstupu, předcházející fáze (vyvolání) se tedy vždy provedou.

Instance InvocationContext prochází třemi fázemi.

3.2.1 Fáze routování

Podle HTTP požadavku je nalezeno první odpovídající pravidlo a je přiřazeno do vlastnosti Route. Pokud žádné neodpovídá, je přiřezeno pravidlo Error404.

Pokud je součástí pravidla továrna na Invocation nebo View, je spuštěna a výsledné instance přiřazeny do příslušných vlastností. Výchozí hodnotou pro Invocation je null, pro View je to výsledek volání základní továrny (ve výchozím stavu nová instance třídy TemplateView).



Obrázek 1: Třída InvocationContext

3.2.2 Fáze vyvolání

Pokud je vlastnost Invocation nastavena fází routování na instanci vyvolání, je toto vyvolání provedeno (viz. sekce o vyvoláních).

3.2.3 Fáze výstupu

V rámci této fáze se provede HTTP odpověď, Pokud bylo nastaveno přesměrování, je odesláno to, jinak se provede vykreslení instance View. Tento proces je podrobněji popsán v samostatné kapitole.

3.2.4 ContinualState

ContinualState, česky spojitý stav, je koncept na pomezí ViewState známého z Webforms a klasické session. Každý ContinualState představuje podobně jako session kolekci typu Dictionary, která umožňuje ukládat dodatečné informace. Tato kolekce ovšem na rozdíl od session není dostupná při každém požadavku, ale je dostupné pouze ve spojité linii požadavků, mezi kterými se předává jako parametr URL.

Tím se podobá ViewState a simuluje tak jeho chování. Ideálním příkladem využití může být například fitrování a seřazení tabulky entit v administraci. Pokud si uživatel otevře dvě okna prohlížeče a v každém si seřadí entity podle jiného atributu, pak při přechodu na druhou stránku mu zůstane v oknech zachováno správné řazení.

Aby se dosáhlo možnosti rozejít se z jedné záložky dvěma směry, je třeba při každém požadavku vytvořit novou kopii ContinualState a vzhledem k tomu, že ContinualState ukládá svá data do session, může tato vlastnost obsadit velké množství paměti. Je proto doplněna kolektorem, který recykluje prostor zabraný nejstaršími instancemi, neboť z praktického hlediska je vždy třeba jen tolik instancí, kolik je otevřeno záložek.

3.3 Routování

Jako routování se označuje činnost, která k URL požadavků přiřazuje vnitřní označení akcí webové aplikace a naopak. Jde tedy o vrstvu abstrakce, která odděluje požadavky uživatele od skutečného uspořádání aplikace.

Základnímu prostředí ASP.NET (resp. frameworku WebForms) v této oblasti donedána chyběly klíčové součásti a URL požadavků tedy musely přímo odpovídat fyzickým souborům aplikace. Tato situace byla napravena příchodem ASP.NET 4.0, které příslušné části doplňuje. Vzhledem k čerstvosti této změny (oficiální vydání dne 12.4.2010) a dalším okolnostem jako jsou náklady na přechod, podpora na hostingu a vydání ASP.NET MVC 2, nedošlo ještě k rozšíření a většina programátorů si této nové možnosti ani není vědoma.

3.3.1 URL rewriting

Předchůdcem routování je URL rewriting (česky přepisování adres), které není přímo součástí aplikace, ale je realizováno jako rozšíření webového serveru. Aktuální verze we-

bového serveri Microsoft IIS společně s rozšířením URL Rewrite podporuje obousměrný přepis pomocí sady pravidel, které programátor definuje v konfiguračním souboru aplikace (Web.config). Přepis požadavků je známý i z jiných webových serverů, například Apache HTTPd s modulem mod_rewrite.

IIS k tomuto přídává ještě přepis odpovědí a tím celou vrstvu odděluje a dává programátorovi možnost zapisovat odkazy v HTML kódu bez ohledu na přepisování. Zjevnou nevýhodou rozšíření URL Rewrite je, že aplikaci je třeba vyvíjet s pomocí IIS místo Development Serveru, který je součástí Visual Studia.

S Development Serverem jsou schopny spolupracovat jiné implementace tohoto konceptu, ale každá z nich má nějakou nevýhodu, což činí routování výhodnějším řešením téměř v každém případě.

Někdy lze ovšem tato řešení kombinovat k dosažení optimálního výsledku. Někdy se stane, že nějakým nedopatřením nebo také častým překlepem uživatelé požadují existující obsah, ale přes špatnou URL. V takovém případě je zbytečné měnit programový kód aplikace a stačí pomocí přepisovacího pravidla tuto URL přesměrovat na korektní.

3.3.2 Struktura a principy routování

Podobně jako u přepisování se i routování skládá z kolekce pravidel. Jedno pravidlo se označuje anglickým termínem route (v českých textech také někdy routa, nebo častěji jen pravidlo routování). Pravidla mohou být různých typů a jednotlivé typy se především liší algoritmem pro rozpoznávání URL v požadavku a analogicky i algoritmem pro vytváření URL.

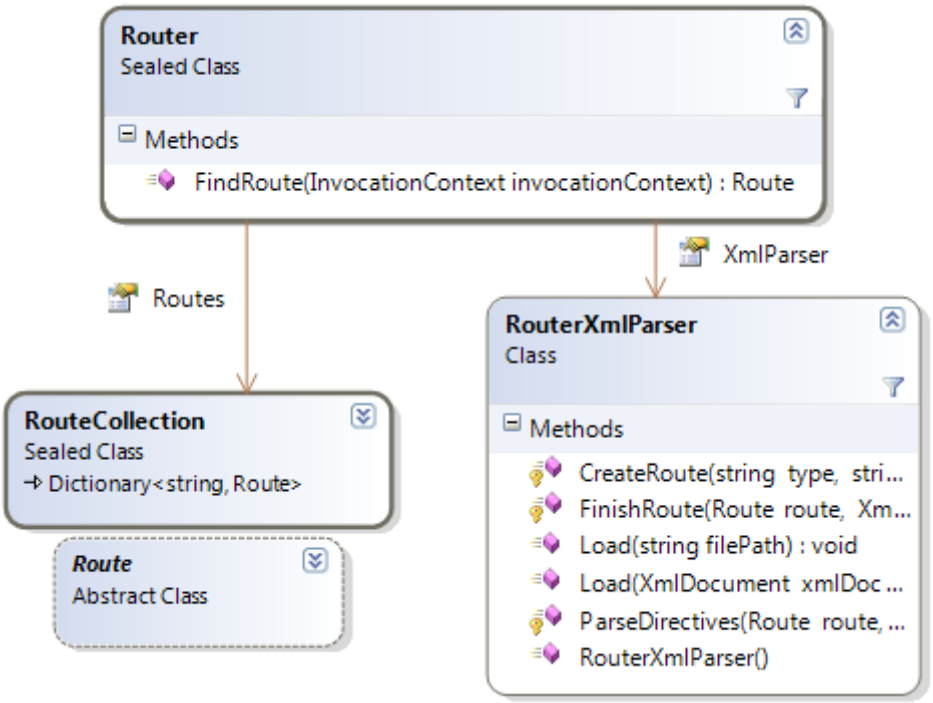
Jednotlivá musí být nějak jednoznačně identifikována, podle implementace se používá buď ručně definovaný identifikátor, nebo kombinace výstupních parametrů (například controller/action v případě frameworku ASP.NET MVC). Na základě této identifikace lze při vytváření URL dohledat příslušné pravidlo a případně zachytit situaci, kdy žádné odpovídající pravidlo neexistuje. Tím se routování významně liší od přepisu URL, neboť zaručuje, že při vývoji aplikace nedošlo k opomenutí.

3.3.3 Účel

Z předchozích odstavců můžeme snadno odvodit hlavní důvody pro zavedení této vrstvy. Prvním důvodem je zaručení správnosti odkazů, které aplikace generuje do HTML kódu. Integrací přímo do aplikace se významně zjednodušuje například řešení odkazu ve více jazycích. Druhým důvodem je zvýšení kvality odkazů z hlediska optimalizace pro vyhledávače. Pokud je z důvodů této optimalizace změnit URL některé stránky, lze to provést na jednom místě v kódu a efekt se rozšíří do celé aplikace.

3.3.4 Návrh a implementace

V pricipech se návrh routování nebude lišit od zavedených zvyklostí. Routování bude součástí počáteční fáze zpracování požadavku a základě jeho výsledku bude určen



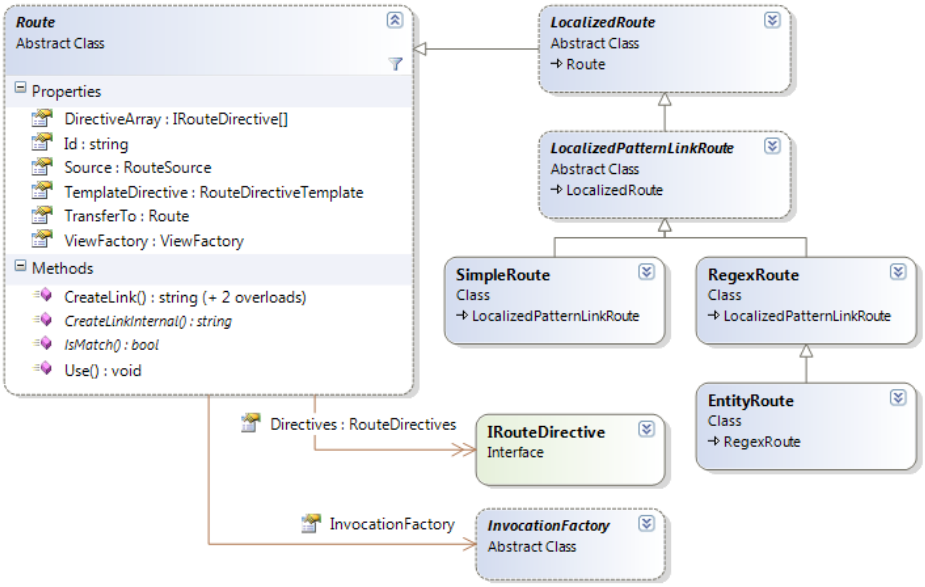
Obrázek 2: Třída Router

konkrétní postup zpracování požadavku. Tento proces spočívá v procházení všech routovacích pravidel a testování, jestli některé odpovídá obdrženému požadavku. Pokud je takové pravidlo nalezeno, je použito k nastavení prostředí, ve kterém se bude požadavek zpracovávat (kontextu vyvolání).

Pokud žádné pravidlo neodpovídá, je řízení předáno záchrannému pravidlu, které zajistí zobrazení hezky vypadajícího hlášení uživateli. Je klíčové, aby toto hlášení na uživatele nepůsobilo jako kritická chyba, a výsledná HTML stránka by měla obsahovat základní navigační prvky společné všem stránkám.

V třídním diagramu můžeme vidět, že kromě metody FindRoute() s popsaným algoritmem je s třídou Router ještě těsně svázána instance třídy RouterXmlParser, která najde uplatnění zejména pro statické stránky a je popsána v samostatné sekci.

Jednou důležitou podmínkou, kterou je třeba, aby mechanismus routování v tomto frameworku fungoval, je korektní práce se statickými stránkami. Statickou stránkou v tomto kontextu není myšlen samostatný HTML soubor, ale takový průběh požadavku



Obrázek 3: Třídní diagram s přehledem dostupných typů pravidel

aplikací, který neprovede žádné vyvolání. V architektuře MVC by takový požadavek tedy používal pouze vrstvu View.

Aby toho bylo dosaženo, je třeba vyvolání chápat jenom jako jednu z možných činností (dále v textu označovaných jako direktivy), kterou lze při požadavku provést. Mezi další možné direktivy lze zařadit použití určité šablony pro výstup nebo přesměrování požadavku na jinou URL. Tyto direktivy jsou součástí definice pravidla routování a jsou provedeny v počáteční fázi vykonávání požadavku.

3.3.4.1 Třídy pravidel routování Všechny třídy pravidel musí být odvozeny od abstraktní třídy Route. Ta poskytuje základ implementace pro všechny potřebné operace s pravidly. Ty lze rozdělit na operace definiční, které budou provedeny pouze jednou, při inicializaci aplikace, a operace opakované.

3.3.4.1.1 Definice pravidel routování Definice pravidla se sestává ze tří částí, které lze provést v rámci jednoho zápisu. Nejprve je třeba vytvořit instanci zavoláním konstruktoru některé neabstraktní třídy, která je potomkem třídy Route. V konstruktoru jsou nastaveny všechny povinné informace jako unikátní identifikátor pravidla a vzor pro srovnávání s URL požadavku.

Další informace se dodávají v rámci druhé fáze a to pomocí nastavování vlastností. Základními vlastnostmi jsou kolekce direktiv a InvocationFactory pro určení vyvolání. Doplňujícími pak Source určující zdroj cesty (definovaná frameworkem, modulem, z XML souboru, nebo ručně), TransferTo pro spojení s jiným pravidlem a ViewFactory pro nastavení generátoru odpovědi. Vyplnění těchto vlastností není povinné, taková stránka bude obsahovat pouze společnou šablonu Layout. Toho lze využít při potřebě dočasně vypnout určitou stránku. Zakomentování celého pravidla by totiž mohlo vést k chybám v aplikaci (pokud by bylo někde toto pravidlo použito pro generování URL).

Pro snadnější definici direktiv poskytuje třída dvě pomocné vlastností - TemplateDirective a DirectiveArray, přičemž u těchto vlastností je možné hodnotu pouze zapisovat. Díky tomu lze použít následující dvě konstrukce:

```
InvocationApplication.Router.Routes.Add(new SimpleRoute("Stranka", "Stranka")
{
    TemplateDirective = new RouteDirectiveTemplate("Pages/Stranka")
});

InvocationApplication.Router.Routes.Add(new SimpleRoute("Presmerovani", "Presmerovani")
{
    DirectiveArray = new IRouteDirective[]
    {
        new RouteDirectiveRedirect("http://www.nahradni-stranky.cz/")
    }
});
```

Nakonec je třeba pravidlo zaregistrovat přídaním do kolekce InvocationApplication.Router.Routes, jak je vidět na předchozím kódu.

3.3.4.1.2 SimpleRoute a RegexRoute SimpleRoute slouží ke spojení právě jedné URL k právě jednomu identifikátoru. Všechny parametry požadavku jsou předávaný jako HTTP parametry (tzn. v URL za otazníkem).

RegexRoute používá jako vzor pro URL regulární výraz a tedy odpovídá více URL. Pomocí regulárního výrazu lze také některé části URL vzít a uložit jako doplňkové parametry k těm získaným běžným způsobem. Naopak při vytváření URL z routovacího pravidla jsou pak předané parametry vloženy přímo do URL, pokud to jde, a zbývající jsou předány běžným způsobem (za otazníkem)

Podrobnější vysvětlení a použití obou těchto tříd je součástí první případové studie.

3.3.4.1.3 EntityRoute EntityRoute je speciální variantou RegexRoute, která nejen z URL převezme parametry, ale také podle nich umožňuje rovnou načíst záznam z databáze. Podrobnější vysvětlení a použití je součástí třetí případové studie.

3.3.5 Speciální routovací pravidla

Framework při inicializaci ověřuje existenci dvou speciálních routovacích pravidel: Error403 a Error404. Tyto pravidla jsou použita tehdy, pokud je uživateli zamítnuto prove-

dení požadavku, resp. pro tento požadavek neexistuje výsledek (například chybějící stránka nebo neexistující aktualita)

Tato pravidla mohou být definována stejně jako ostatní pravidla, ale je doporučeno pro ně použít SimpleRoute a doplnit jim pouze šablonu pro vykreslení.

3.3.6 Router.xml

Soubor Router.xml je umístěn v adresáři App_Data/ a umožňuje vytváření routovacích pravidel bez nutnosti kompilování programového kódu[2]. Je v něm možno definovat pouze pravidla typu SimpleRoute a RegexRoute.

Obsah souboru může vypadat například takto:

```
<?xml version="1.0" encoding="utf-8" ?>
<Router>
    <SimpleRoute Id="Error404" Pattern="Error404" TemplateSource="Pages/Error404" />
    <SimpleRoute Id="Error403" Pattern="Error403" TemplateSource="Pages/Error403" />
    <SimpleRoute Id="Index" Pattern="" TemplateSource="Pages/${locale}/Index" />
    <SimpleRoute Id="Kontakt" Pattern="[[@url]]" TemplateSource="Pages/${locale}/Kontakt" />
    <RegexRoute Id="Clanek" Pattern="Clanek/(?&lt;Id&gt;\d+)" TemplateSource="Pages/Clanek" /
        >
</Router>
```

3.4 Vyvolání

Vyvolání (třídy odvozené od třídy Invocation) jsou základním stavebním kamenem aplikace založené na tomto frameworku a je v nich umístěna část aplikační logiky. V architektuře MVC jde tedy o spojení vrstev Model a Controller. Na rozdíl od MVC jsou ale vyvolání konceptuálně více odstíněna od HTTP požadavku.

Vzhledem k tomu, že programátora aplikace nelze prakticky od HTTP požadavku oddělit, bylo třeba vytvořit takové řešení, které by mu tak významným způsobem pomohlo, že by se snažil použití třídy HttpContext vyhnout z vlastní iniciativy.

3.4.1 Parametry a výsledek

Tímto řešením je použít silné typování pro parametry i výsledky vyvolání. Parametry jsou přitom myšlena zpracovaná vstupní data z HTTP požadavku a výsledky jsou data, která budou následně předána vrstvě View.

Tento koncept si můžeme předvést na krátkém příkladu. Předpokládejme, že vytváříme akci pro zobrazení statistik hráčů šachu v určitý rok. Na vstupu tedy potřebujeme znát jméno hráče a rok, na výstupu pak počet výher, počet proher, součet bodů a navíc ještě oblíbenou barvu figurek.

```
public class MyTestInvocationParameters : InvocationParametersBase
{
    public string PlayerName = "";
```

[2]Podrobnější vysvětlení tohoto souboru je součástí první případové studie.

```
        public int Year = DateTime.Now.Year;

        public ChessPlayer Player
        {
            set
            {
                if (value == null)
                    PlayerName = "";
                else
                    PlayerName = value.Name;
            }
        }
    }

    public enum PreferredSetColor { Black, White };

    public class MyTestInvocationResult : InvocationResultBase
    {
        public int Wins { get; set; }
        public int Losses;
        public int Points;
        public PreferredSetColor Color;
    }
```

Třída *Parameters musí implementovat rozhraní IInvocationParameters (například skrz třídu InvocationParametersBase) a dále pro ni platí omezení, že všechna data, které mají být čteny z HTTP požadavku, musí být pole (ne vlastnosti). Jinak řečeno vlastnosti lze nastavovat pouze kódem. Toho lze využít například při vytváření odkazů.

Místo toho, abychom věděli, že toto vyvolání pracuje se jménem hráče a předávali tento parametr, přeneseme zodpovědnost za výběr konkrétní vlastnosti hráče šachu na implementaci vyvolání. Pokud tedy budeme vytvářet odkaz na statistiky aktuálního mistra světa, můžeme použít následující konstrukci:

```
string url = myInvocation.CreateLink(new MyTestInvocationParameters() { Player =
    GetWorldMaster() });
```

Třída *Result nemá žádná omezení kromě povinnosti implementovat rozhraní IInvocationResult (nebo být odvozena od InvocationResultBase).

3.4.2 Stav

Protože by podle návrhu koncepce instance Invocation neměla obsahovat pracovní data, ale existuje potřeba mít tyto data někde uložena pro snazší rozdělení výkonného kódu do metod. Proto kromě třídy pro parametr a výsledek používá vyvolání také třídu pro stav provádění.

Framework s instancí této třídy nijak nepracuje a pro udržení čistoty kódu by to neměl mimo konkrétní vyvolání dělat ani programátor.

3.4.3 Třída vyvolání

Při definici třídy vyvolání je třeba ji odvodit od generické abstraktní třídy Invocation a uvést jednotlivé třídy, které reprezentují parametry, výsledek a stav vyvolání. Pokud vyvolání nemá vlastní třídu pro některou z těchto účelů, je možné uvést třídu Invocation*Base.

```
public class MyTestInvocation
    : Invocation<MyTestInvocationParameters, MyTestInvocationResult, InvocationStateBase>
{
    public int MySetting { get; set; }

    public MyTestInvocation()
    {
        MySetting = 10;
    }

    [InvocationPrepare]
    public InvocationCheckStatus CheckInvocation()
    {
        return InvocationCheckStatus.Enabled;
    }

    [InvocationCheck(Order = 1000)]
    public InvocationCheckStatus CheckInvocation()
    {
        return InvocationCheckStatus.Enabled;
    }

    [InvocationCheck(Order = 2000)]
    public InvocationCheckStatus CheckInvocation()
    {
        if (Parameters.Year < 1990)
            return InvocationCheckStatus.Disabled;
        else
            return InvocationCheckStatus.Unchecked;
    }

    protected ChessPlayer FindPlayer()
    {
        return Persistence<ChessPlayer>.FindOne<IUserEntity>(Expression.Eq("Name",
            Parameters.PlayerName));
    }

    [InvocationExecute]
    public void ExecuteInvocation()
    {
        ChessPlayer player = FindPlayer();
        if (player != null)
        {
            Result.Wins = player.GetWinsInYear(Parameters.Year);
            Result.Losses = player.GetLossesInYear(Parameters.Year);
            Result.Color = player.PreferredSetColor;
```

```
                Result.Points = (Result.Wins − Result.Losses) ∗ MySetting;
            }
        }
    }
```

Vlastnosti vyvolání představují jeho nastavení nezávislé na požadavku uživatele. V našem případě mámě nastavení MySetting, podle kterého se počítají v naší aplikaci body (pro jednoduchost je zde uveden pouze jeden koeficient, v praxi by jich bylo samozřejmě více).

3.4.3.1 Metody a životní cyklus Metody vyvolání lze rozdělit do čtyř skupin:

1. pomocné
2. přípravné
3. ověřovací
4. vykonávací

Pomocné jsou pouze běžné metody, které framework nijak nepoužívá. V našem příkladu je to metoda FindPlayer().

Ostatní skupiny jsou vždy označeny speciálním atributem (InvocationPrepare, InvocationCheck a InvocationExecute), přičemž se provádějí v příslušné skupině vždy postupně podle vlastnosti Order tohoto atributu (výchozí je 0, řazení je vzestupné). Provádění celé skupiny ale může být zastaveno výjimkou. Výhození výjimky v kterékoliv části provádění má pak za následek ukončení celého vyvolání a uživateli je zobrazeno pouze chybové hlášení. Uživatelé tedy neuvidí výpis výjimky, což je nejen uživatelsky příjemnější, ale také bezpečnější.

Přípravné a ověřovací úzce spolupracují. Úkolem přípravných metod je provést pouze tolik činností, aby mohly ověřovací metody provést svoji činnost. Ověřovací činosti pak na základě parametrů a dat získaných přípravnými metodami vrátí jednu z hodnot výčtu InvocationCheckStatus. Hodnota Enabled povolí uživateli přístup, hodnota Disabled ho zakáže a Unchecked stav nezmění.

Pokud bychom v našem příkladu měli řešena přístupová práva, pak by přípravná metoda mohla nejdříve načíst do stavu vyvolání hráče podle parametrů a ověřovací metody by následně zkontrolovala, jestli má uživatel přístup k informacím tohoto hráče.

Pokud je výsledkem ověření hodnota Enabled, tak je provedena i poslední skupina metod - metody vykonávací. Jejich úkolem je naplnit výsledek vyvolání daty.

3.4.3.2 Přidání routovacího pravidla Aby mohl uživatel toto vyvolání skutečně vyvolat, je třeba k němu přidat routovací pravidlo. To provedeme v rámci metody InitializeApplicationInvocations() třídy aplikace. Definice je stejná jako u pravidel bez vyvolání, jenom je třeba doplnit továrnu na naše vyvolání.

Továrnou je generická třída InvocationFactory, jejíž generický parametr nastavíme na třídu, kterou chceme továrnou vytvářet. InvocationFactory umožňuje také přidat zpětná

volání (Workers), která se provedou při vytváření vyvolání. Takto přidaný kód slouží ke změně výchozího nastavení vlastností vyvolání (viz. výše).

```
InvocationApplication.Routes.Add(new SimpleRoute("MyTest", "MojeStatistiky/Hrac")
{
    InvocationFactory = new InvocationFactory<MyTestInvocation>(),
    TemplateDirective = new RouteDirectiveTemplate("MyTest")
});
```

3.4.3.2.1 Vytváření odkazů Pro vytváření odkazů na vyvolání existuje zkratka. Metoda CreateLink() přebírá jako parametr přímo instanci třídy, která je uvedena jako třída parametrů a tímto zjednodušuje použití a zpřehledňuje kód, neboť lze opět využít výhod silného typování.

3.5 Moduly

Modul je třída odvozená od třídy Module a slouží jako kontejner pro továrny vyvolání a také k nastavení těchto továren a k nim příslušných pravidel routování. K tomu slouží metody InitializeInvocations() a InitializeRouter(), které jsou ve třídě Module označeny jako abstraktní. Tyto metody jsou vyvolány v rámci stejnojmenných metod při inicializaci aplikace a tedy i ve stejném pořadí (nejdříve *Invocations, pak *Router).

3.5.1 Praktický příklad

Tento příklad navazuje na příklad v sekci o vyvoláních. Tentokrát je ovšem testovací vyvolání součástí modulu. InitializeInvocations() i InitializeRouter() obsahuje velmi podobný kód s tím rozdílem, že je vyvolání i pravidlo přiřazeno k modulu.

```
public class MyModule : Module
{
    public InvocationFactory<MyTestInvocation> MyTestInvocationFactory { get; protected set; }

    public MyModule()
        : base("My")
    {
    }

    public override void  InitializeInvocations ()
    {
        MyTestInvocationFactory = new InvocationFactory<MyTestInvocation>()
        {
            Workers = new InvocationFactory<MyTestInvocation>.WorkerDelegate(delegate(
                MyTestInvocation invocation)
            {
                invocation.Module = this;
                invocation.MySetting = 42;
            })
        };
    }
```

```
    public override void InitializeRouter ()
    {
        InvocationApplication.Routes.Add(new SimpleRoute(Id + "/MyTest", "[[@url]]")
        {
            Source = RouteSource.Module,
            InvocationFactory = MyTestInvocationFactory,
            TemplateDirective = new RouteDirectiveTemplate(Id + "/MyTest")
        });
    }
}
```

3.5.2 Registrace modulu

Použité moduly programátor musí registrovat v metodě InitializeApplicationModules() třídy InvocationHttpApplication následujícím způsobem:

```
public override void InitializeApplicationModules()
{
    InvocationApplication.Modules.RegisterModule(new UserModule());
}
```

3.5.3 Administrace

Administrační modul je takový modul, který je odvozen od třídy AdministrationModule a od obyčeného modulu se liší tím, že při jakémkoliv vyvolání, které je součástí tohoto modulu, je nejprve ověřeno, že je uživatel přihlášen. Pokud není, tak je přesměrován na stránku přihlášení do administrace.

Navíc má ještě abstraktní metodu GenerateAdministrationMenu(), která slouží k poskládání obsahu hlavního menu administrace. Tuto metodu tedy musí autor modulu doplnit, ale může ji také vytvořit prázdou. V takovém případě modul nebude dostupný z hlavního menu, ale může být například volán z jiného modulu.

3.5.3.1 CrudAdministrationModule CrudAdministrationModule je třída odvozená od AdministrationModule a obsahuje základní činnosti pro administraci záznamů v databázi. Zároveň také obsahuje obecnou ukázkovou implementaci metody GenerateAdministrationMenu()

```
public override void GenerateAdministrationMenu(AdministrationMenu menu)
{
    menu.GetGroup(I18N.Translate("/Modules/" + Id + "/MenuGroup", "Default"))
        .Add(Id, new MenuItem(I18N.Translate("/Modules/" + Id + "/MenuItemTitle", Id),
            ListEntitiesInvocationFactory.Route));
}
```

Příklad použití CrudAdministrationModule je v třetí případové studii, kde je implementován modul novinek.

3.6 Ovlivnění mezi moduly

Použití třídy InvocationFactory pro nastavování parametrů vyvolání poskytuje každému modulu možnost upravit nastavení vyvolání v jiném modulu připojením dalšího kódu do řetězu vyvolání. Modul tak může zjistit, zda je přítomný jiný modul a upravit jeho vlastnosti a vyvolání. Použít lze samozřejmě jakékoliv vlastnosti tříd modulů, tj. je možné používat podmínky typu „existuje-li modul s rozhraním IAdministrationModule“.

Aby byla ovšem zajištěna základní inicializace vyvolání, je třeba, aby moduly tyto změny prováděly pouze v rámci své metody InitializeForeignInvocations(). Tato metoda přebírá jeden parametr, celé číslo v rozsahu 0 až 1000, které určuje krok provádění. To probíhá tak, že se nejprve se vyvolá pro všechny moduly InitializeForeignInvocations(0), následně pro všechny moduly InitializeForeignInvocations(1), atd. Tím lze zajistit základní návaznost mezi změnami v případě ovlivnění mezi třemi a více moduly.

Pro pokročilejší propojení modulů lze použít návrhový vzor Inversion of Control nebo aspektově orientované programování, například pomocí frameworku Spring.NET (http://www.springframework.net/). V takovém případě je třeba použít Inversion of Control kontejner v metodě InitializeApplicationModules() v souboru Global.asax.

3.7 Logování

Logování, neboli proces zaznamenávání informací o běhu aplikace, je velmi užitečné zejména při doplňování nového kódu do kódu již existujícího (zejména v případě, že neznáme jeho přesný průběh). Proto tento vnitřní činnosti frameworku generují informace o průběhu požadavku, podobně jako to dělá základní ASP.NET.

Framework nepoužívá vlastní řešení, ale využívá open-source knihovnu Apache log4net (http://logging.apache.org/log4net/index.html), která je součástí dlouhotrvajícího a komplexního projektu zabývajícího se tímto tématem. Díky tomu je toto řešení velmi pružné a existuje řada nástrojů, které lze pro analýzu logu využít.

Jedním z vedlejších cílů frameworku je přinutit programátora, aby udržoval celou aplikaci v konzistentním stavu. Proto je většina selhání řešena výjimkami a do logu jsou zapisovány pouze problémy s nízkou pravděpodobností, že zůstanou nepovšimnuty nebo že vzniknou zásahem neprogramátora do běžící aplikace - především se jedná o selhání související s lokalizací.

4 Vrstva View

V tradičním návrhovém vzoru MVC představuje vrstva View oddělenou část aplikace, která zajišťuje pouze generování samotné odpovědi na požadavek, tedy formování výstupu z dat, která obdrží od vrstvy Controller. Přestože tento framework MVC architekturálně neodpovídá, přebírá z něj částečně tento koncept.

4.1 Úvod

Při srovnání například s frameworkem ASP.NET MVC má tento framework podstatně obecnější architekturu vrstvy View a drží se více základního návrhu vzoru MVC. Pomocí dalších vrstev umožňuje realizovat výstup libovolném (binárním či textovém formátu) a zároveň dodává abstrakci, která umožňuje zpracovávat šablony výstupu nezávisle na poskytovateli (například .ascx UserControl nebo C# třída).

4.2 Třída View

Třídy odvozené od třídy View slouží k nastavení chování výstupu aplikace, tj. určují obsah poslaný v odpovědi. Každý požadavek musí mít právě jednu aktivní instanci třídy View a ta je uložena ve vlastnosti View příslušné instance InvocationContext. V textu bude dále vlastnost InvocationContext.Current.View, tedy aktivní instance třídy View pro právě zpracovávaný požadavek, označována jako View nebo aktivní View.

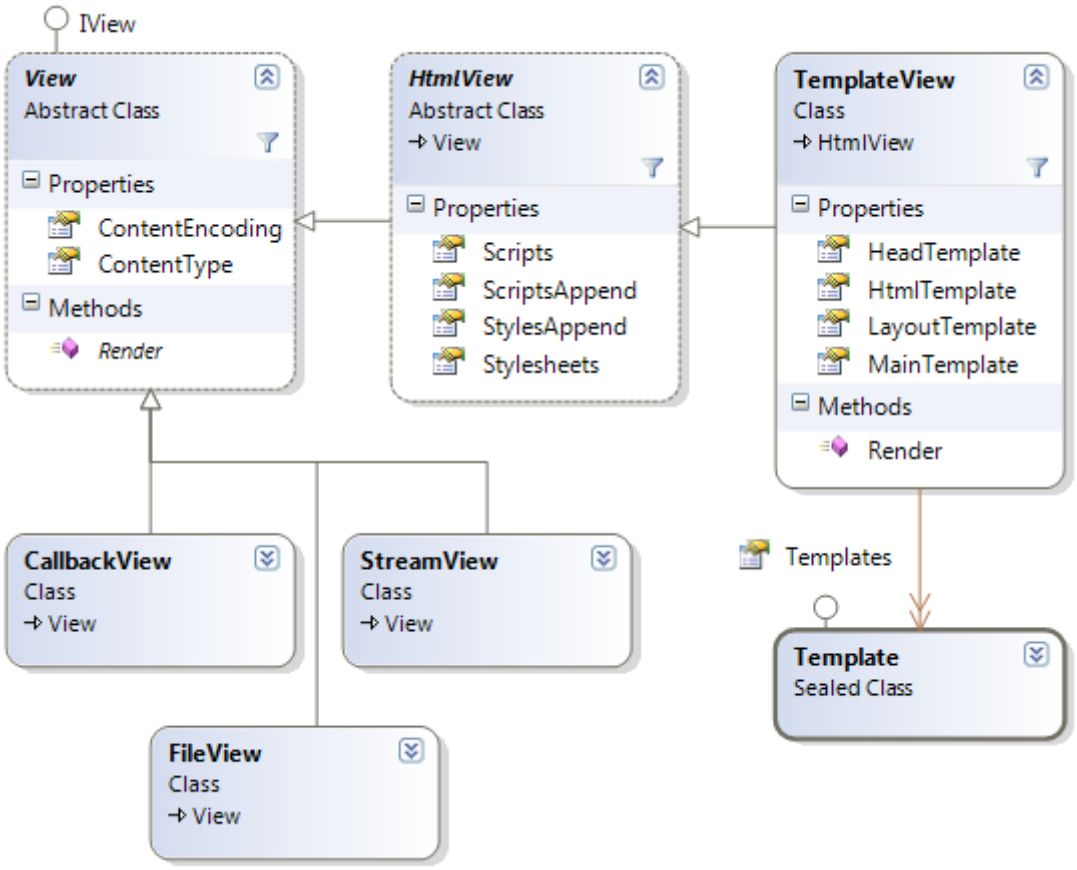
K vytvoření této instance slouží abstraktní třída ViewFactory. Její konkrétní odvozené třídy musí implementovat metodu Create() vracející instanci třídy View. Framework poskytuje dvě základní továrny:

- ApplicationViewFactory, která pro vytvoření zavolá metodu CreateView() instance InvocationHttpApplication
- CallbackViewFactory, kterou může ve spojení s delegátem programátor použít pro vlastní nastavení

4.2.1 Výchozí chování

Výchozím chováním je použití ApplicationViewFactory, přičemž výchozí implementace v InvocationHttpApplication vytvoří TemplateView s nastavenými šablonami Html, Head a Layout na jejich výchozí cesty (viz. dále). V případě, že je v kontextu nastavena továrna vyvolání a toto vyvolání implementuje rozhraní IAdministrationInvocation, je místo výchozí šablony Layout použita šablona Administration/Layout.

Z toho tedy vyplývá, že pro úplnost stránky je třeba ještě doplnit hlavní šablonu. Tu musí poskytnout buď pravidlo routování, nebo vyvolání.



Obrázek 4: Třídní diagram hierarchie tříd View

4.2.2 Určení chování v Route

Jak bylo popsáno už v sekci o routování, lze pomocí nastavení vlastnosti ViewFactory resp. přidáním direktivy typu RouteDirectiveTemplate definovat speciální View, resp. hlavní šablonu. V případě určení hlavní šablony se direktiva provede pouze pokud je aktivní View instancí TemplateView.

4.2.3 Určení chování v Invocation

Použitou instanci View a hlavní šablonu lze změnit také pomocí vyvolání. Toto nastavení má přednost přede všemi ostatními. Změnu lze provést tím, že bude vyvolání implementovat příslušné z následujících rozhraní. Skrze rozhraní IViewFactoryInvocation vyvolání poskytuje odkaz na instanci ViewFactory, skrze rozhraní IMainTemplateInvocation přímo instanci hlavní šablony. Opět platí, že pokud je hodnota null, tak ke změně nedochází a určení chování je tiše ignorováno. V případě určení hlavní šablony opět platí, že k němu dochází jenom pokud je aktivní View instancí TemplateView, jinak je tiše ignorováno.

```
public interface IViewFactoryInvocation
{
    ViewFactory ViewFactory { get; }
}

public interface IMainTemplateInvocation
{
    Template MainTemplate { get; }
}
```

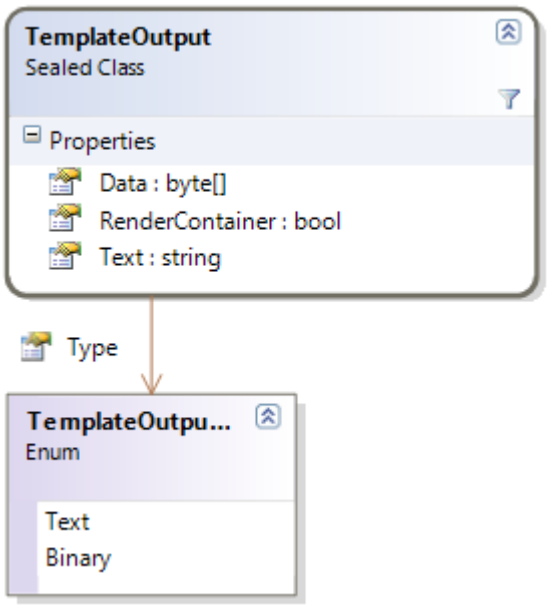
4.3 Šablony

Šablony slouží k formátování dat do požadovaného tvaru. Z vnějšího pohledu je vstupem šablony její označení a kolekce dat k zformátování. Výstupem šablony mohou být binární data, nebo častěji text.

Z vnitřního pohledu může být šablonou programový kód (v některém z jazyků podporovaných platformou .NET), soubor, nebo cokoliv jiného. Vnitřní pohled je dám použitým poskytovatelem vykreslení.

Vykreslení šablony se provádí metodou Render() a má čtyři fáze:

1. Spojení data předaných parametrem s daty přímo v instanci šablony.
2. Zpracování zpětných volání pro cache výstupu.
3. Pokud nebyla použita cache, je provedena detekce vhodného poskytovatele vykreslení a jeho spuštění.
4. Pokud je výsledek (obdržený z cache nebo vykreslením) textový, jsou něj jsou aplikovány doplňující úpravy, například obalení HTML elementem div.



Obrázek 5: Třída TemplateOutput

4.3.1 Výsledek vykreslení

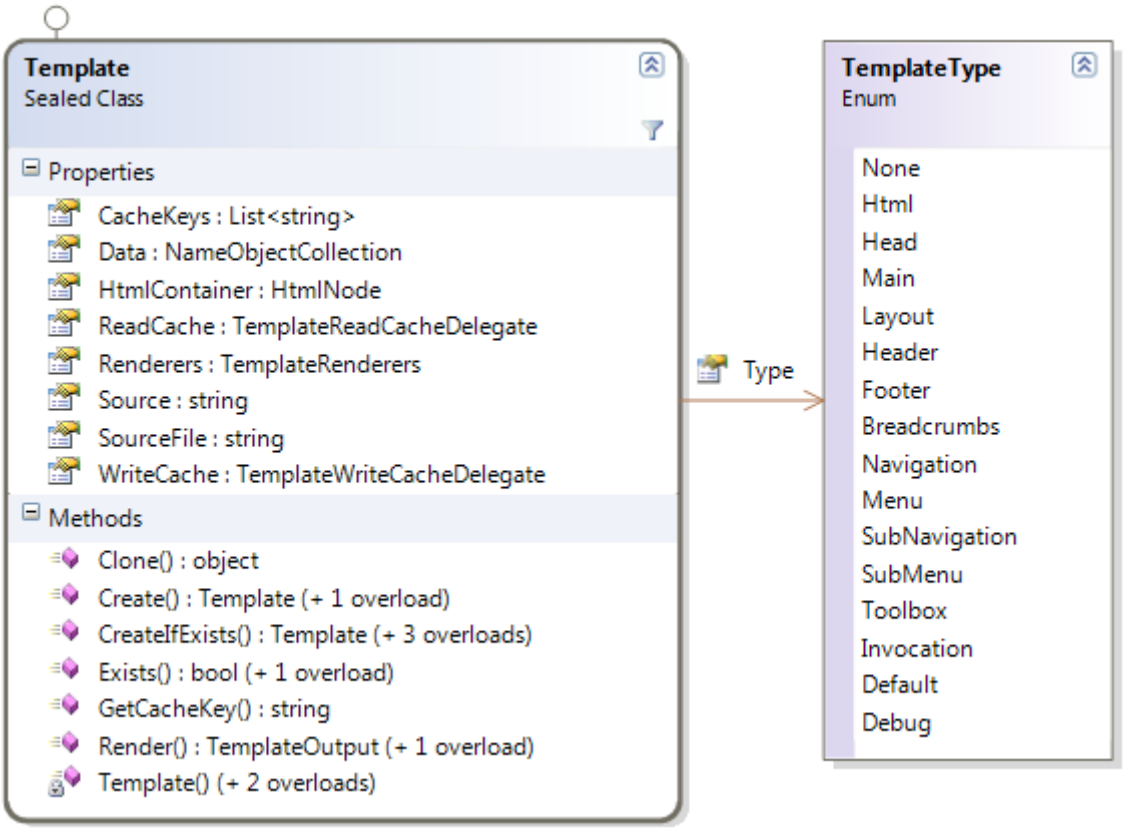
Výsledkem metody Render() je instance třídy TemplateOutput. Ta poskytuje univezální reprezentaci jak pro textový, tak pro binární výstup. O kterou z těchto dvou variant se jedná určuje hodnota vlastnosti Type. V případě textového výstupu se pracuje s vlastností Text, v případě binárního s vlastností Data.

Pro textový výstup lze pomocí vlastnosti RenderContainer ovlivnit, zda bude výstup ještě obalen doplňujícím HTML elementem div. Ten slouží k zjednodušení stylování, protože obsahuje označení šablony a také počitadlo, kolikrát byla šablona vykreslena během aktuálního požadavku.

4.3.2 Třída Template

Třída Template je uzavřená a slouží jako vrstva abstrakce a spojovací článek mezi vlastním kódem programátora a poskytovatelem vykreslení.

4.3.2.1 Vlastnosti Nejdůležitější vlastností je Source, jejíž hodnota je zároveň jediným parametrem konstruktoru. Její hodnota je řetězec identifikující šablonu. Ve vlastnosti



Obrázek 6: Třída Template

Source lze používat zástupný řetězec „$locale“, který se při zpracování nahradí kódem jazyka aktuálně zpracovávaného požadavku. Upravenou hodnotu lze získat z vlastnosti SourceFile.

Další důležitou vlastností je kolekce Data, jejíž obsah je při zpracování šablony spojen s parametrem data. Při tomto spojení mají hodnoty uložené v kolekci předané jako parametr přednost.

Doplňujícími vlastnostmi jsou Type a HtmlContainer. Tyle slouží k určení účelu šablony při automatizovaném vykreslování rozvržení stránky. HtmlContainer je předlohou pro obalující element div.

Vlastnosti ReadCache a WriteCache jsou zpětná volání určená pro cache šablon. Jejích použití je popsáno v kapitole zabývající se implementací cache v tomto frameworku. Pokud je využit navržený vzor pro řešení cache šablon, je ke generování klíče položky cache použita metoda GetCacheKey(). Ta vytváří klíč kombinací vlastnosti SourceFile a těch párů klíč-hodnota z předaná kolekce dat, jejichž klíče jsou uvedeny v kolekci CacheKeys. Kolekce CacheKeys je ve výchozím stavu prázdná.

4.3.2.2 Metody Alternativou ke konstruktoru je statická metoda Create(), která přebírá stejné parametry jako konstruktor a vždy vrací instanci Template nebo skončí výjimkou. Statická metoda CreateIfExists() navíc přebírá jako nepovinný parametr kolekci dat, kterou následně použije k otestování, zda je některý poskytovatelů vykreslení schopen šablonu s tímto označením a daty vykreslit. Pokud není, je vrácena hodnota null.

Související je instanční metoda Exists(), která vrací hodnotu bool v závislosti na tom, jestli je možné šablonu s danými parametry vykreslit.

Metoda Render() provádí vykreslení a byla popsána v textu výše. Metoda Clone() provádí hluboké klonování.

4.3.3 Umístění souborů

Většina výchozích poskytovatelů vykreslení (viz. dále) je založena na souborech. Soubory s šablonami pro tyto poskytovatele je třeba umístit do adresáře Templates/ v kořenovém adresáři aplikace. Pro poskytovatele vykreslení založeného na třídách je výhodné umístit soubory s kódem tamtéž, neboť bude mít programátor větší přehled o dostupných šablonách, ale především vývojové prostředí zařadí šablonu automaticky do korektního namespace, tj. NazevMehoProjektu.Templates.

4.3.4 Poskytovatelé vykreslování

Pro větší pružnost je ve frameworku přidána vrstva poskytovatelů vykreslování. Jedná se o instance tříd odvozených od TemplateRenderer, které jsou zaregistrovány do kolekce Template.Renderers.

Každý poskytovatel musí implementovat dvě metody s následujícími signaturami:

```
public bool Detect(Template template, NameObjectCollection data);
public TemplateOutput Render(Template template, NameObjectCollection data);
```

Metoda Detect() slouží k ověření, zda je poskytovatel najít konkrétní instanci (třídu, soubor, ...) odpovídací šabloně se zadanými parametry. Metoda Render() pak pro již ověřenou šablonu vytvoří instanci TemplateOutput s výstupem.

4.3.4.1 ClassTemplateRenderer Tento poskytovatel vykreslování je určen pro integraci logiky šablon do kompilovaného kódu. Je vhodné jej použít pro binární výstup nebo pokud je třeba vytvořit knihovnu šablon.

Aby byl schopen najít potřebné třídy, je nutné jej korektně nastavit. Ve výchozím stavu prohledává assembly PoskiNET a assembly aplikace. Pro přidání dalšího assembly s šablonami je třeba rozšířit jeho nastavení následujícím kódem v metodě InitializeApplicationView() v Global.asax:

```
var renderer = Template.Renderers["ClassTemplateRenderer"];
renderer.ClassPrefixes.Add("MojeSablony.Templates.");
renderer.ExploreAssembly(typeof(MojeSablona).Assembly);
```

Při odvození šablony od základní třídy ClassTemplate je třeba implementovat metodu RenderTemplateOutput(), která musí vracet instanci třídy TemplateOutput. Tuto práci si může programátor zjednodušit odvozením TextClassTemplate nebo BinaryClassTemplate, kde stačí používat instanci TextWriter (resp. BinaryWriter) a to v implementaci metody Render().

Od každé z těchto tří tříd ještě existuje rozšířená generická varianta Invocation*ClassTemplate, kterou lze svázat přímo příslušnou třídou odvozenou od Invocation. Pokud bude při vykreslování této šablony aktuální vyvolání v kontextu odpovídat tomuto typu v generickém parametru, bude tato instance vyvolání dostupná vlastnost Invocation a tedy umožní silně typovaný přístup k vlastnostem a výsledku vyvolání.

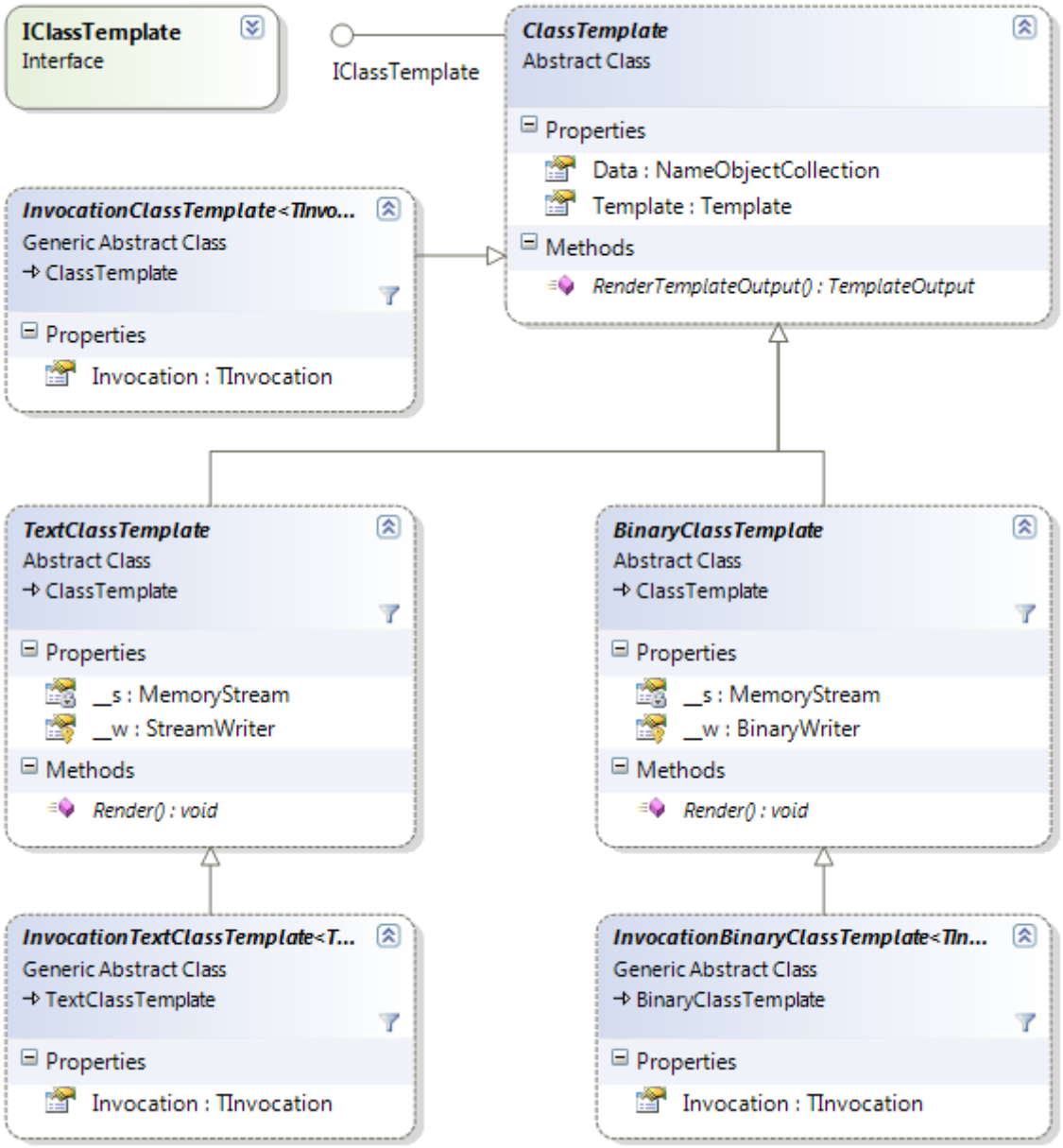
4.3.4.2 HtmlTemplateRenderer HtmlTemplateRenderer je poskytovatel vykreslování založený na souborech. K vlastnosti Source přidává „html“ a hledá soubor s tímto názvem v adresáři šablon. Výsledný soubor beze změn pošle na výstup.

4.3.4.3 AscxTemplateRenderer AscxTemplateRenderer je poskytovatel umožňující vykreslit soubory .ascx podobným způsobem jako v základním ASP.NET. Je založen na souborech, podobně jako HtmlTemplateRenderer, s tím rozdílem, že přidává koncovku „ascx“. Pro tyto soubory platí stejná pravidla jako ve Webforms.

Tento poskytovatel vykreslování je určen především pro programátory, kteří již dříve pracovali s Webforms nebo ASP.NET MVC. Pro začínající programátory je vhodnější použít následujícího poskytovatele, SparkTemplateRenderer.

4.3.4.4 SparkTemplateRenderer SparkTemplateRenderer poskytuje vykreslování skrze Spark View Engine (http://sparkviewengine.com/). Ten nabízí pokročilou syntaxi se zápisem přátelským pro HTML kodéry. Proto je doporučeno jeho použití místo AscxTemplateRenderer nebo ClassTemplateRenderer, pokud není závažný k opaku.

SparkTemplateRenderer je založen na souborech a používá koncovku „spark“.



Obrázek 7: Třídní diagram hierarchie tříd ClassTemplate

Lepší představu o výhodách Spark View Engine poskytne následující krátká ukázka:

```
<viewdata products="IEnumerable[[Product]]"/>
<ul if="products.Any()">
  <li each="var p in products">${p.Name}</li>
</ul>
<else>
  <p>No products available</p>
</else>
```

4.3.5 Základní šablony

Framework poskytuje několik základních šablon pro TemplateView. Jedná se o Html a Head. Obě tyto šablony jsou parametrizovány přes kolekci Flags v kontextu vyvolání a zohledňují kolekce, které TemplateView dědí od HtmlView.

Tyto základní šablony jsou implementovány jako třídy, aby nebylo třeba je přenášet jako samostatné soubory. Vytvořením vlastních šablon se stejným názvem mohou být tyto šablony nahrazeny.

4.3.5.1 Layout Specifická je šablona Layout, u které výchozí nastavení předpokladá existenci, ale neposkytuje ji, protože by měla obsahovat základní rozvržení specifické pro konkrétní projekt.

Bez ohledu na poskytovatele, kterého programátor zvolí, je ve výsledku vykreslení nahrazen text „<PoskiNET:MainTemplate />“ výsledkem vykreslení šablony Main. Pokud šablona Main existuje, ale Layout neobsahuje tento text, bude vykreslení Main také vyvoláno.

5 Lokalizace a internacionalizace

Pro dobré pochopení této kapitoly je třeba si nejdříve objasnit pojmy lokalizace a internacionalizace. Internacionalizace, často též označovaná zkratkou I18N (kde 18 reprezentuje příslušný počet písmen uprostřed anglického termínů internationalization), je postup či sada programových nástrojů, kterými je software připraven tak, aby podporoval více jazyků. Lokalizace (též L10N) představuje následné doplnění software o podporu konkrétního jazyka. Společně se tyto pojmy někdy shrnují pod pojem globalizace.

Z hlediska webové aplikace představuje globalizace znatelně méně činností, než u klasických aplikací. Přesto však je třeba řešit například tyto oblasti:

- formátování čísel (například anglická desetinná tečka a česká desetinná čárka) a peněžních částek
- formátování datumu a času, časové zóny
- překlady textů
- smysluplnost některých údajů (například IČ může být povinné při registraci v české mutaci webové aplikace, ale nemá smysl pro anglickou mutaci)

V závislosti na komplexnosti projektu a zvoleném aplikačním prostředí musí programátor vytvořit vlastní řešení pro jeden nebo více z těchto požadavků. Například v prostředí PHP nemůže programátor s pomocí samotného prostředí počítat vůbec, zatímco prostředí .NET s sebou přináší rozsáhlou podporu. Díky tomu zůstává vyřešit hlavně různé jazykové verze textů, které aplikace vypisuje.

V této kapitole se budeme věnovat pouze textům, které definuje programátor při vývoji aplikace. Další, samostatnou částí, je lokalizace uživatelských dat. Ta sice s tímto tématem souvisí z vnějšího pohledu, požadavky na řešení v této oblasti jsou však složitější, a proto je výhodnější oba problémy řešit odděleně.

Stanovme si tedy základní požadavky na cílové řešení. Základní, ne však úplně zřejmý, je požadavek na evidenci všech jazyků, které aplikace podporuje. Tato potřeba nemusí nutně existovat u interních webových aplikací jako jsou informační systémy, ale z hlediska webové prezentace je důležité znát, jaký jazyk obsahu je dostupný a jaké volby tedy nabídnout uživateli (nejčastěji se tato nabídka realizuje formou vlajek států v některé z okrajových oblastí stránky).

Druhý, už zřejmější, požadavek je samotné přeložení textů - podsystém řešící lokalizaci by měl být pro kombinaci jazyka a nějakého klíče, vyhledat příslušný přeložený text.

Třetí a čtvrtý požadavek úzce souvisí. V zájmu vytvoření co nejkvalitnějšího díla, je třeba mít možnost sledovat a sestavit seznam dosud nepřeložených textů. Tyto je pak třeba převádět mezi přirozeným formátem překladových souborů a formátem, který může zpracovat profesionální překladatel (myšlena například osoba překládající z češtiny do španělštiny apod.).

Posledním, pátým, požadavkem pak je snadná udržovatelnost přirozeným formátem překladových souborů pro případ malých změn. Naprosto bežným požadavkem, se

kterým se lze potkat v praxi, je potřeba klienta doplnit nebo opravit jeden překlad - například nadpis stránky. Tato činnost by neměla být závislá na programátorovi, či speciálních nástrojích.

Stručně tedy můžeme tyto požadavky shrnout následovně:

1. seznam podporovaných jazyků
2. překlad textů
3. sledování nepřeložených textů
4. podpora pro profesionální překlad
5. jednoduchost drobných změn

Z těchto pěti je první sice důležitý, ale vytvořit vlastní implementaci takového mechanismu je jednoduché a pokud bude existovat optimální řešení pro zbývající požadavky, pak lze takové řešení snadno rozšířit o tento bod. Druhý požadavek je naprosto základní a celý problém definuje. V konečném důsledku má tedy smysl se zabývat pouze zbývajícími třemi body a případnými vlastnostmi, které jednotlivá řešení přinášejí navíc.

5.1 Přehled dostupných řešení

Abychom zjistili, které z dostupných řešení nejvíce odpovídá našim požadavků, provedeme u každého stručný rozbor vnitřních pricipů, ukázku použití a nakonec jakým způsobem jednotlivé požadavky řeší.

5.1.1 Lokalizace v kódu

V tomto přehledu nelze opominout nejjednodušší metodu a to uložení lokalizovaných textů přímo do kódu. Přestože se může zdát toto řešení jako krajně nevhodné, můžeme v některých případech najít dostatečné opodstatnění pro jeho použití. Například pokud by měl být dosah těchto překladů omezen pouze jednu či dvě stránky (kontaktní formulář). Jako obecné řešení to však použít nelze a z dlouhodobého hlediska může čas na údržbu výrazně růst.

```
public enum Jazyk { Cesky, Anglicky };

public class Preklad
{
    public readonly string USER;

    public Preklad(Jazyk j)
    {
        if (j == Jazyk.Cesky)
            USER = "Uzivatel";
        if (j == Jazyk.Anglicky)
            USER = "User";
    }
```

```
}

Label1.Text = (new Preklad(aktivniJazyk)).USER;
```

1. seznam jazyků řeší a je pevně dán v kódu
2. překlad textů řeší, jako bonusovou vlastnost přináší součinnost s IntelliSense nápovědou
3. nepřeložené texty lze snadno najít například debuggerem, bez dalších rozšíření řešení se obtížně hledají v běžící prezentaci
4. seznam klíčů k přeložení lze vytvořit snadno textovým editorem, vložení hotových překladů zpět je mírně komplikovanější
5. je třeba kompilovat kód, podle případu i celé aplikace - drobné úpravy jsou tedy komplikované

5.1.2 Resources

Použití resources k lokalizaci textů bychom mohli označit za přirozenou metodu prostředí ASP.NET [3]. Z hlediska programátora jde o řešení nejlépe prozkoumané a podporované, má tedy širokou podporu navzdory některým slabým stránkám. Mezi ně lze patří především obtížná strukturovatelnost a pro lokalizaci nevhodný formát souboru.

Resources původně sloužily především k vkládání externích souborů přímo do spustitelného souboru aplikace. Tímto způsobem jsou například řešeny ikony .exe aplikací pod Microsoft Windows. V prostředí .NET je pak tento koncept pozměněn na formu XML serializace, součástí resource souboru tak mohou být instance libovolné serializovatelné třídy, mezi jinými právě třídy String nebo Icon. Je tedy na programátorovi, aby dobře oddělil lokalizované texty od ostatních resources.

Resources se dělí na globální, které jsou přístupné v celé aplikaci, a lokální, které přisluší pouze k jedné stránce, tedy .aspx souboru. V obou případech je znát silná integrace do prostředí, neboť z globální resources jsou automaticky generované silně typované obaly, které jako v předchozím řešení umožňují snadno pracovat s požadovaným řetězcem. Lokální resources se zase implicitně přiřazují do vlastností komponent na stránce, pokud je jejich klíč zapsán v určitém formátu.

Nakonec je třeba ještě poznamenat, že je možné používat resources jak jako externí soubory (pokud jsou jednom z adresářů App_GlobalResources nebo App_LocalResources) nebo je lze tradičním způsobem přikompilovat přímo do výsledného assembly (při umístění v jiném adresáři). Pro účely lokalizace je samozřejmě žádoucí první z těchto variant.

Celkově se resources jeví jako velice dobrá volba, byť zbytečně komplexní pro účely lokalizace.

```
// silne typovany pristup
Label1.Text = Resources.Texty.User
```

```
// pristup pres funkci
Label1.Text = (String)GetGlobalResourceObject("Texty", "User");
```

1. podporované jazyky neřeší a nelze je přímo zjistit, lze však prozkoumat adresář, kde jsou resources uloženy
2. integrace do prostředí .NET toto řešení zvýhodňuje
3. výchozí hodnota pro nepřeložený text null, podle použití může být buď neviditelná nebo vyvolat výjimku související s použitím null objektu
4. existuje jak nástroje pro překladatele pracující přímo s resource soubory (http://resxtranslator.codeplex.com/), tak nástroje pro konverzi z a do formátu přirozenějšího pro překladadatele (http://resx.sourceforge.net/)
5. drobné úpravy jsou možné, ale vyžadují zvýšenou opatrnost, protože chybějící resource soubor (například důsledkem překlepu) k vyvolání výjimky při zobrazení stránky

5.1.2.1 gettext Knihovna gettext vznikla za účelem lokalizace programů napsaných v jazyce C a spouštěných pod unixovým operačním systémem. Existuje implementace pro prostředí .NET, která umožňuje použití zdrojových souborů připravených pro gettext jako resources souborů. Použití je tedy z zásadě stejné a i pro tento formát existuje řada nástrojů pomáhající v překladu.

Existuje zde však několik rozdílů. Překladové soubory většinou definují překlad skutečné fráze výchozího jazyka na frázi zvoleného jazyka, například:

```
msgid "My name is %s."
msgstr "Moje jméno je %s."
```

Není tedy jasně definovaný neměnný klíč, ale v případě jakékoliv změny původní věty je třeba doplnit nový překlad. V spojení s tím, že je třeba před použitím přeložit tyto překlady do formátu resources, je celý proces více pracný a také náchylný k chybám, než při přímém použití resources. Toto jinak běžné řešení je tedy pro webové aplikace v ASP.NET nevhodné.

5.1.3 XLIFF

XLIFF [4] je zkratka pro XML Localization Interchange File Format. Již z názvu je tedy znát, že tento formát by navržen především pro výměnu překladů a je vhodný spíše pro překlad dokumentů, než přímo lokalizaci.

Původní dokument je nejdříve rozdělen na nelokalizovatelnou kostru (v případě webové prezentace by zde byly HTML značky) a XLIFF soubor. Pomocí některého z dostupných nástrojů je následně XLIFF soubor přeložen profesionálním překladatelem a

společně s kostrou integrován do nového, přeloženého dokumentu. Tento postup se také někdy označuje jako Extract-Localize-Merge [5].

Navzdory tomuto zaměření je formát XLIFF používán k lokalizaci například v PHP frameworku Symfony [6]. Vzhledem k tomu, že není momentálně dostupná implementace použití tohoto formátu pro .NET, není možné toto řešení přímo použít - může být ale základem pro řešení vlastní.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xliff PUBLIC "\textit{-}XLIFFDTD XLIFF//EN"
  " http: // www.oasis-open.org/committees/xliff/documents/xliff.dtd">
< xliff  version="1.0">
  <file  source-language="EN" target-language="fr" datatype="plaintext"
       original ="messages" date="2008-12-14T12:11:22Z"
      product-name="messages">
      <trans-unit>
          <source xml:lang="en-US">We are testing XLIFF.</source>
          <target state="translated"  xml:lang="fr">Nous testons XLIFF.</target>
      </trans-unit>
  </file >
</ xliff >
```

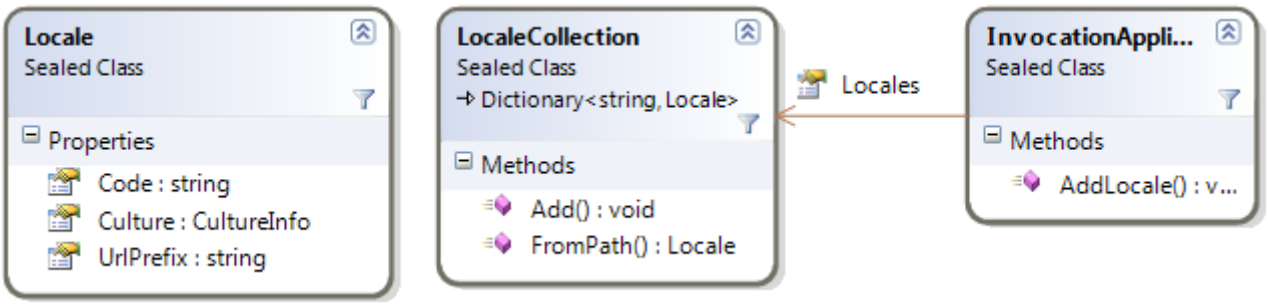
5.1.4 PoskiPHP

Na základě podobných požadavků byl pro framework PoskiPHP zvolen jednoduchý formát souborů čistého text, kde každý neprázdný řádek reprezentuje jeden překladový záznam ve formátu klíč, jeden a více tabulátorů, přeložený text. Podobně jako u resources obsahuje aplikace více překladových souborů. Tyto jsou pro snadnou manipulaci umístěny v adresáři s názvem dle cílového jazyka.

Překlad klíče se provádí pomocí funkce podobně jako u gettextu, s tím rozdílem, že existuje nepovinný druhý parametr, který specifikuje požadovaný soubor překladu. Pokud příslušný překlad neexistuje, propadne překlad do výchozího překladového souboru. Pokud není nalezen ani tam, je pro snadnější hledání nepřeložených textů vrácen klíč (tedy první parametr).

```
echo __('submitted', 'users');
```

1. seznam dostupných jazyků je řešen formou jednoduchého seznamu
2. překlad textu umožňuje jednu úroveň záchrany / propadnutí (anglicky fallback)
3. požadavky na nepřeložené texty lze logovat, ve výsledné HTML stránce jsou nepřeložené texty zobrazeny jako klíče
4. překladové soubory jsou bez újmy na funkčnosti upravitelne v běžném tabulkovém kalkulátoru (například MS Excel nebo OO Calc)
5. úprava překladového souboru je velmi jednoduchá, jen je třeba dát pozor, aby editor nenahradil oddělovač tabulátor mezerami (nebo jej nahradit alternativním oddělovačem „=“)



Obrázek 8: Třídní diagram seznamu podporovaných jazyků

5.2 Návrh a implementace

V předchozí části jsme odhalili, že existuje několik zavedených řešení lokalizace textů, ale také, že pro účely webových prezentací v ASP.NET není ani jedno optimální. Můžeme ale z těchto řešení vyjít a vytvořit vlastní, které bude splňovat požadovaných pět bodů. Zároveň budeme moci toto řešení úzce zaměřit a tím i snížit komplexnost a zjednodušit praktické použití.

5.2.1 Seznam podporovaných jazyků

Při implementování této vlastnosti můžeme využít existující prostředků .NET frameworku, konkrétně třídy CultureInfo. Pro účely spojené s routováním a persistencí ale potřebujeme připojit ještě vlastní označení jazyka a také prefix URL, který se pro tento jazyk bude používat. Tím nám vznikne třída Locale.

Tyto třídy stačí zabalit do kolekce, kterou připojíme jako statickou vlastnost třídy InvocationApplication, která uchovává všechny objekty společné pro celou aplikaci. Tím zajistíme možnost iterace přes všechny podporované jazyky aplikace.

Výchozí implementace třídy InvocationHttpApplication definuje čestinu jako výchozí jazyk. Nahrazením metody InitializeApplicationLocales() lze toto změnit. Při přidávání definic je vhodné použít dvouznakové jazykové kódy podle normy ISO 639-2 [7]. Je také nutné, aby právě jeden z přídaných jazyků měl prázdný URL prefix a jeho definice byla uvedena jako první. Pro korektní formátování je dále požadováno, aby předaná instance CultureInfo popisovala specifickou kulturu.

```
public override void InitializeApplicationLocales ()
{
    InvocationApplication.AddLocale("cs", "", new System.Globalization.CultureInfo("cs-CZ"));
    InvocationApplication.AddLocale("en", "en/", new System.Globalization.CultureInfo("en-US"));
}
```

5.2.2 Formát souboru překladů

Formát soubor s překlady textů musí být kompromisem mezi jednoduchostí úpravy a komplexností struktury, kterou v něm lze vyjádřit. Pro tento účel je nejvhodnější založit vlastní formát na XML. XML je dnes velmi rozšířený a jeho pricipy snadno pochopitelné, lze tedy předpokládat, že samotná forma zápisu nebude pro neprogramátory problematická. Zároveň je možno využít mechanismů DTD nebo XML Schema pro validaci struktury a tím se dále pojistit proti náhodným chybám při drobných úpravách.

Tyto soubory je třeba umístit do podadresáře jazyka v adresáři Translations/ v kořenovém adresáři aplikace. Například překladový modulu User pro jazyk angličtina (předpokládejme, že příslušná instance Locale byla vytvořena s kódem „en“) by měl mít cestu ~/Translations/en/User.xml.

Aby byla dosážena větší pružnosti při znovupoužití existujících překladů, je formát definován tak, že svou strukturou připomíná strom, přičemž jednotlivá přeložená položka je označena anglickým termínem message, který se většinou v této souvislosti používá, a skupina, ve které mohou být tyto položky a další podskupiny sdruženy je označena jako namespace kvůli podobnosti s názvovými prostory v jazyce C# jak v zápisu i tak i fungování. Oddělovačem úrovní je zvoleno dopředné lomítko.

Každou přeloženou položku je možné vyjádřit kombinací klíčů názvových prostorů a svého vlastního klíče. Pravidla jsou zde podobná jako v adresářové struktuře operačních systémů typu unix. Kořenový názvový prostor je označen jedním lomítkem („/“) a všechny ostatní názvové prostory musí být vloženy v nějakém nadřazeném názvovém prostoru. Identifikátor, nebo také klíč, názvového prostoru musí vždy končit lomítkem a být vepsán v atributu key příslušného XML elementu namespace.

Pro zjednodušení zápisu a větší znovupoužitelnost definic překladů je možno v klíči názvového prostoru použít relativní cestu a to vynecháním lomítka na začátku klíče. Klíč se při načítání doplní klíčem nadřazeného elementu, přičemž kořenový element má implicitní a nezměnitelný klíč odpovídající kořenovému názvovému prostoru. Dále je možné vepsat několik úrovní názvového prostoru do jednoho klíče, tedy například key=„/Forms/Validators/“.

Zápis překladových položek je proti názvovým prostorům jednodušší. Klíč položky nemůže obsahovat dopředná lomítka a je vždy relativní ke svému přímému rodičovskému elementu namespace. Hodnotu překladu lze zapsat dvěma způsoby. První je zápsání elementu message jako nepárového a vyplnění atributu value, který má při zpracování přednost. V tomto případě je třeba vždy dbát korektního použití XML entit (například pro ostré závorky nebo ampersand). Druhý způsob spočívá v zapsání elementu jako párového a vložení hodnoty do jeho obsahu. Zde je možno se vyhnout přepisu znaků na XML entity vložením hodnoty uvnitř sekce CDATA. Proto je tento postup doporučovaný pro zápis překladů obsahujících HTML značky.

V rámci překladů lze používat také syntaxi metody TranslateInString(), která bude popsána dále. Smyslem tohoto zápisu je umožnit spojování překladů.

Následuje stručná ukázka složená z částí základního překladového modulu Common.

```
<?xml version="1.0" encoding="utf-8" ?>
```

```
<namespaces>
    <namespace key="/Common/">
        <message key="Back" value="zpět" />
        <message key="ExceptionMessage" value="Nastala␣výjimka␣{0}:␣{1}" />

        <namespace key="Locales/">
            <message key="cs" value="Čeština" />
            <message key="en" value="Angličtina" />
        </namespace>

        <message key="Test">
            <[CDATA[Jazyky jsou {{Locales/cs}} a {{/Common/Locales/en}}.]]>
        </message>
    </namespace>

    <namespace key="/Forms/">
        <message key="SubmitLabel" value="Odeslat" />
    </namespace>

    <namespace key="/Forms/Validators/">
        <namespace key="Castle/">
            <message key="NonEmpty" value="Pole␣'${label}'␣je␣vyžadováno." />
        </namespace>
        <message key="Unique" value="Pole␣'${label}'␣musí␣mít␣unikátní␣hodnotu." />
        <message key="Integer" value="Pole␣'${label}'␣musí␣být␣celé␣číslo." />
    </namespace>
</namespaces>
```

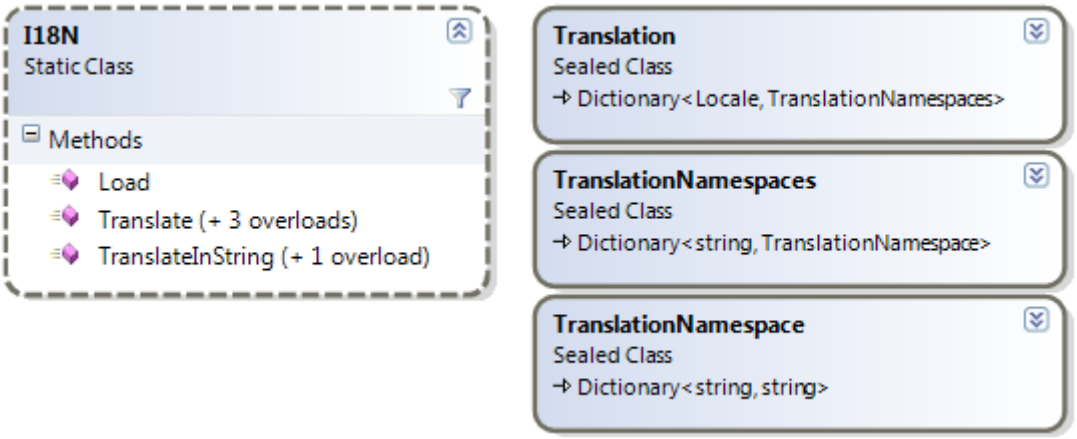
5.2.3 Třída I18N

Pro práci s překlady má programátor aplikace k dispozici statickou třídu I18N a její tři základní metody včetně několika přetížení pro přehlednější zápis. Jejich nejširší signatury jsou následující:

```
class I18N
{
    public static void Load(string module);
    public static string Translate(Locale locale, string input, string fallback);
    public static string TranslateInString(Locale locale, string input);
}
```

Metoda Load() nemá žádné přetížení a slouží k načtení a zpracování překladového souboru předaného jako jediný parametr. Tento parametr je pouze identifikátor modulu, bez přípony a bez cesty. Překladové soubory Common, Administration a Meta jsou nahrány frameworkem automaticky. Soubory příslušející k modulům jsou nahrány výchozí implementací metody InitializeTranslations() modulu. Ostatní soubory musí programátor aplikace nahrát ručně nahrazením metody InitializeApplicationTranslations() ve třídě InvocationHttpApplication (resp. souboru Global.asax).

Metoda Translate() slouží k přeložení klíče input do jazyka locale, přičemž tento jazyk musí být definovaný. Parametr locale je nepovinný a pokud nebude předán, tak



Obrázek 9: Třídní diagram překladů

se použije jazyk aktuálního kontextu vyvolání. Pokud není příslušný překlad nalezen, bude vrácena hodnota fallback. Pokud tato hodnota není předána, je vrácen plný klíč překladu. Tím je možné při vývoji snadno v prohlížeči či v HTML kódu stránky najít nepřeložené klíče.

Metoda TranslateInString() je doplňková a je užitečná například při aplikaci na jednoúčelové mikrošablony. V parametru input provádí nahrazení všech podřetězců odpovídajících formátu XYZ výsledkem volání Translate(„XYZ“). Parametr locale je opět nepovinný.

Pro obě překladové metody platí, že hodnota input nesmí být null.

Zde je několik praktických příkladů pro použití metod Translate() a TranslateIn-String():

```
__w.Write(I18N.Translate("/Invocations/" + invocation.Id + "/Title"));

string navigation = string.Format(I18N.Translate("/Common/YouAreHere"), breadcrumbs.Render())
    ;

string localizedLink = I18N.TranslateInString(this.Link);
```

6 Formuláře

6.1 Úvod

Jak bylo popsáno v předchozích kapitolách, komunikace mezi uživatelem a webovou aplikací probíhá výhradně přes cyklus požadavků a odpovědí. Tedy veškeré informace, které uživatel chce předat webové aplikaci musí zakódovat do svého požadavku. Jednu základní informaci, totiž kterou stránku požaduje v odpovědi, popisuje v URL, kterou požadavek začíná. Další informace, především takové, které nemohou být předem známé (jako například jméno či adresu), musí k požadavku připojit v takovém formátu, aby . Takový postup by byl však pro uživatel nepřehledný a velmi těžko pochopitelný, a proto je třeba tuto činnost usnadnit tím, že mu poskytneme nabídku možných informací, které je aplikace ochotna přijmout. Metodou pro předání této nabídky jsou webové formuláře.

6.2 Teorie

Podobně jako jejich papírové protějšky, poskytují webové formuláře pole, která svým tvarem a chováním pomáhají uživateli formulovat předávané informace ve správném tvaru. Proto se můžeme ve webových formulářích vyskytují společné základní prvky. Mezi tyto prvky patří především jednořádková pole pro vyplnění textu nebo čísla v určité maximální délce, víceřádková pole pro volný text, zatržítka, výbery, a možnost připojit přílohu.

Aby uživatel předával co nejvíce relevatní data, bývají formuláře opatřeny kontrolním mechanismem, bežně označovaným výrazem validace. Tu lze rozdělit podle dvou kritérií - podle obecnosti a podle místa zpracování.

Podle obecnosti je nejčastější syntaktická validace, tedy ověření, jestli mají data požadovanou formu. Typickým příkladem je ověření, že je konkrétní pole vyplněno, nebo že odpovídá požadovanému formátu (jde o celé číslo, má přesně 8 číslic, atd.). Schopnosti syntaktické validace jsou omezené a díky tomu můžeme najít společné vzory, které lze následně snadno automatizovat.

Sémantické ověření dat, tedy ověření významu, bývá méně časté a je většinou přenášeno na člověka, který obdržená data následně zpracovává. Variant je zde v podstatě neomezeně nejvhodnější kombinaci pravidel je třeba určit vždy pro každý případ individuálně. Tato pravidla bývají zároveň mnohem komplikovanější než pravidla syntaktické validace a jejich realizace vyžaduje hlubší zamyšlení nad možnými okolnostmi a důsledky. I zde však můžeme najít podobnosti mezi nejčastěji se vyskytujícími pravidly. Například vzor „zadaná hodnota se vyskytuje určité množině“ lze realizovat jako „IČ firmy je v rejstříku firem“ nebo „vyplněné přihlašovací jméno je v seznamu uživatelů s povoleným přístupem“. Další běžnější vzory mohou být zaměřeny na porovnávání datumů nebo kontrolu platnosti zvolene kombinace hodnota. Díky těmto vzorům lze automatizovat také některé konkrétnější případy sématické validace.

Podle místa zpracování rozpoznáváme validaci na straně serveru a validaci na straně klienta, kde klientem je myšlen webový prohlížeč uživatele. Podle zásad bezpečnosti we-

bových aplikací by měla být vždy implementována alespoň validace na straně serveru, jelikož její proběhnutí lze zaručit. Naopak validaci na straně klienta může uživatel velice snadno vyřadit, a proto by měla sloužit pouze k tomu, aby uživateli přijemnějším způsobem sdělila to, co by se dozvěděl z chybových hlášení po odeslání požadavku s nevalidními daty.

Díky tomu, že se jedná o jediný praktický způsob přenosu informací od uživatel k webové aplikaci, jsou formuláře základním stavebním kamenem informačních systémů, jako například aplikací pro řízení workflow (WMS) nebo pro řízení vztahu se zákazníky (CRM). V těchto aplikacích, jejichž základním posláním je organizovat data přijatá od uživatelů, je co nejlépe zpracovaná validace a použitelnost formulářů, kritickým prvkem, a je těmto tématům věnována velká část času vývoje aplikace.

Jiný přístup ovšem převládá v oblasti webových prezentací. Protože je cílem zajistit především přesun informací směrem k uživateli, bývají formuláře často zanedbávány a při vývoj časově podhodnoceny. To pak může výrazně ovlivnit konečný cíl webové prezentace, tedy generování obchodních příležitostí.

Ovšem právě proto, že není možné se formulářům dostatečně věnovat, potřebuje vývojář mít k dispozici takový nástroj, který mu v co nejkratším čase umožní implementovat formulář blížící se co nejvíce optimálnímu stavu. Optimální stav je takový, kdy uživatel vyplnil údaje přesně podle představy provozovatele daného formuláře (resp. webové prezentace) a tím podpořil jeho byznys. Jako ideální variantu si lze představit lidského operátora, který uživatele při vyplňování formuláře vede. Proto je třeba, aby mohl vývojář snadno realizovat následující:

1. Pohodlně a přehledně definovat strukturu formuláře včetně možnosti tvorby vlastních prvků.
2. Použít běžné validační vzory a mít možnost definovat vlastní.
3. Abstrahovat od technických podrobností souvisejících s pricipy fungování webových aplikací.
4. Definovat automatické opravy přijatých hodnot.
5. Seskupovat a umisťovat jednotlivé prvky formuláře.
6. Jednotným a přehledným způsobem řešit zobrazení chybových hlášení v případě špatně vyplněných hodnot.
7. Doplňovat návody k vyplnění jednotlivých prvků formuláře.

6.2.1 XForms

K běžným webovým formulářům vytvořeným pomocí kombinace (X)HTML a klientských skriptů vznikl v průběhu minulého desetiletí standard XForms [8]. Je založen na XML a jeho účelem bylo popsat nejen způsob, jakým bude data uživatel

vyplňovat, ale také především pravidla validace a vztahy vyplňovaných dat k existujícím datům.

Díky tomu, že tento formát řeší problematiku komplexně a s velkým záběrem, tak v současné době žádný z rozšířených prohlížečů ve výchozím stavu neposkytuje implementaci (viz. seznam dostupných implementací na http://www.w3.org/MarkUp/Forms/wiki/XForms_Implementations). Z toho důvodu zatím je tento, jinak velmi zajímavý, formát na okraji zájmu a jeho nasazení nelze v několika příštích letech očekávat, zejména v oblasti webových prezentací, kde je hlavním cílem oslovení do nejširší skupiny uživatelů

6.3 Dostupná řešení

Při práci s formuláři v prostředí ASP.NET se nám dnes nabízejí v zásadě tři možnosti. První z nich je obecná a nízkoúrovňová a díky tomu je použitelná bez ohledu na použitý framework. Pro získání dat zadaných uživatelem použijeme přímo parametry předané v instanci HttpRequest a zobrazení formuláře uživateli je řešeno čistým HTML kódem. Přestože toto řešení nachází běžně uplatnění pro formuláře malého významu či rozsahu, spolehlivou aplikaci na něm lze postavit jen těžko.

Zbývající dvě možnosti jsou dány protichůdnými principy, které reprezentují dva základní směry ve vývoji webových aplikací. Událostmi řízené programování frameworku Webforms a paradigma MVC, které odděluje vrstvu prezentace formuláře od zpracování přijatých dat.

6.3.1 Webforms, použití komponent

Framework Webforms nabízí integrovaný přístup, kdy je formulář složen z komponent, přičemž každá komponenta se kompletně stará o jednu či několik hodnot poskytovaných uživatelem.

Komponenty přistupují k surovým datům z HTTP požadavku a od této části programátora odstiňují. Není tedy na první pohled zřejmé, kolik hodnot konkrétní komponenta obsahuje a ani to není žádoucí. Programátorovi jsou tyto hodnoty prezentovány už přeložené jako běžné vlastnosti komponeny, podobně jako vlastnosti určující, jakým způsobem se bude uživateli prezentovat žádost o vložení dat.

Definice komponent nejběžněji probíhá pomocí značkovacího jazyka přímo v .aspx části stránky. Díky generátoru je pak tato komponenta k dispozici v kódu jako bežné třídní pole.

```
E−mail:
<asp:TextBox ID="Email" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="RequiredFieldValidatorEmail"
    runat="server" ErrorMessage="RequiredFieldValidator"
    ControlToValidate="Email"></asp:RequiredFieldValidator>
```

```
protected void Page_Load(object sender, EventArgs e)
{
    if (IsPostBack && IsValid)
```

```
        PridejPrijemce(Email.Text);
}
```

6.3.2 ASP.MVC

Ačkoliv je ASP.MVC jen jedním zástupcem z několika volně dostupných MVC frameworků pro ASP.NET, můžeme jej použít jako ideální reprezentativní vzor, neboť paradigma oddělení vrstev Model-View-Controller vede často k implementaci velmi podobných řešení nejen pro ASP.NET, ale také v ostatních prostředích jako Java, PHP nebo Ruby.

Na formuláři se postupně podílí všechny tři vrstvy aplikace takovým způsobem, že samotná data jsou uzavřena ve vrstvě Model a popis vzhledu je oddělen do vrstvy View.

V následujícím příkladu můžeme vidět toto rozdělení v praxi. Je zde uvedena ukázka jedné z nejběžněji implementovaných formulářových akcí, konkrétně akce Upravit, v anglické terminologii označovaná jako Update.

6.3.2.1 Model Modelem může být libovolný objekt, ale ve většině případů jím bývá přímo business entitita. Podobnost zde můžeme najít například s řešením formulářů v MVC frameworcích nad jazykem Java, kde modelem bývá obvykle instance speciální třídy, která se označuje jako form bean.

Kromě kontejneru na obdržená uživatelská data je model také zodpověný za validaci těchto dat. Tu může programátor buď řešit ručně, nebo využít možností, které mu poskytují další knihovny. Zde je například pro model použita knihovna Castle.ActiveRecord včetně validátorů.

```
public class Clovek : ActiveRecordValidationBase<Clovek>
{
    public int Id;

    [ValidateNonEmpty]
    public string Jmeno;
}
```

6.3.2.2 Controller Vrstva Controller musí při tomto rozdělení provést tři základní kroky:

1. data získaná z formuláře od uživatele přenést do modelu
2. validovat data a případně je zpracovat (uložit, poslat e-mailem, ...)
3. přenést data modelu do vrstvy View

Pro implementaci těchto kroků je obvykle vhodné využít pomocných metod poskytovaných modelem.

```
[ControllerAction]
public void Upravit(int id)
{
    Clovek c = Clovek.FindById(id);
    // nacteni uzivatelskeho vstupu
    c.UpdateFrom(Request.Form);
    // validace
    if (c.IsValid())
        c.Save();
    // prenos do View
    ViewData["Clovek"] = c;
}
```

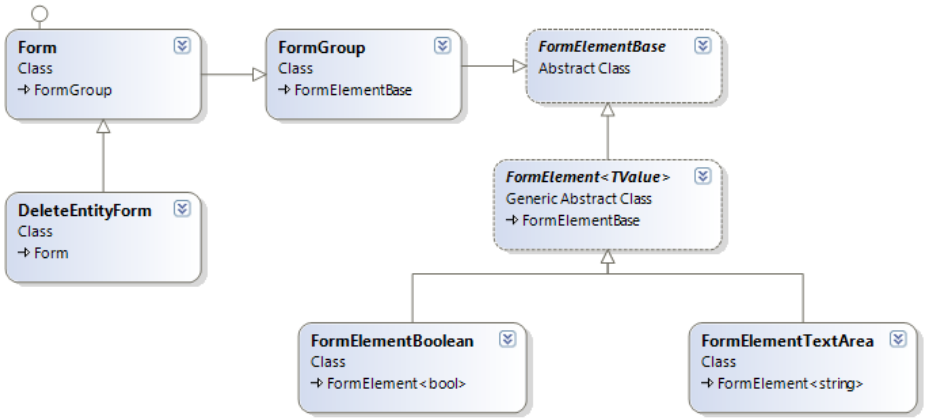
6.3.2.3 View Na vrstvě View pak spočívá samotné zobrazení formuláře uživateli. Výhoda tohoto rozdělení spočívá v tom, že samotný formulář může být řešen mnoha různými způsoby a je jen třeba, aby data odeslaná prohlížečem byla následně čitelná pro controller při jejich zpracování. Například hodnocení v rozsahu 1 až 5 může uživatel zadávat jako číslo do textového pole o délce jednoho znaku, výběřem z pěti označených přepínačů, nebo javascriptovou komponentou zobrazující možnosti jako hvězdičky.

Přímým důsledkem této variability v možnostech vstupu je přenesení části validace do vrstvy Vie, které si můžeme ukázat opět na příkladu hodnocení. Zatímco v případě přepínačů se lze spolehnout, že běžným použitím uživatel nepřenese nevalidní data, tak v případě textového pole musíme ověřit, že uživatel zadal čislici a nikoliv písmeno a v případě javascriptové komponenty bývá třeba ověřit, že na straně uživatele nenastala nějaká chyba v běhu skriptu.

Toto řešení ovšem přináší i nevýhodu, tedy že oddělení v kódu programu často nutí programátora duplikovat validaci na dvě nezávislá místa. Pokud tedy vznikne potřeba rozšiřit validační pravidla modelu, je třeba tato pravidla doplnit do vrstvy View všude tam, kde se v příslušejícím controlleru provádí validace tohoto modelu. V opačném případě by uživatel obdržel nanejvýše informaci o selhání validace, ale již by se nedozvěděl, které pole a jak má upravit.

Řešení tohoto problému framework ASP.NET MVC přenechává na programátorovi. Jedno z jich je rozšíření validace modelu tak, aby poskytovala nejen logickou hodnotu, zda jsou data validní či nevalidní, ale také textovou informaci popisující důvody případně nevalidity. Tyto informace následně controller přenese mezi vrstvami.

```
<form action="<%= Url.Action(new { Action="Upravit", ID=ViewData.Clovek.Id }) %>" method="
    post">
    Jméno:
    <%= Html.TextBox("Jmeno", ViewData.Clovek.Jmeno) %>
    <%= Validation.NotEmpty("Jmeno") %>
    <input type="submit" value="Uložit" />
</form>
```



Obrázek 10: Hierarchie tříd formuláře

6.4 Návrh a implementace vlastního řešení

Abych se co nejvíce přiblížil potřebám webových prezentací, vyšel jsem ve svém řešení z principů Webforms. Formulář je v tomto pohledu stromem komponent založeným na návrhovém vzoru kompozit. V těchto třídách jsou pak integrovány všechny potřebné vlastnosti a funkce, od načítání dat z požadavku, přes jejich validaci až po vykreslení formuláře do HTML.

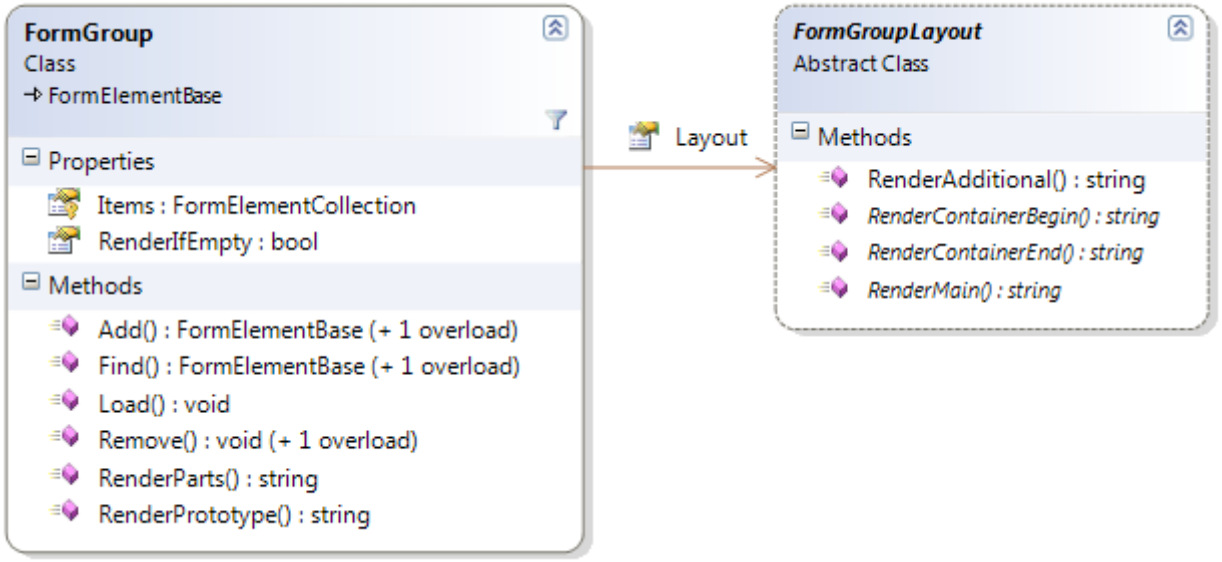
Existují tři základní úrovně: formulářů (třída Form a odvozené), skupin (třída FormGroup) a položek/elementů (třída FormElementBase). Položky jsou vždy listy stromu, formulář vždy kořenem. V doporučené struktuře je mezi kořenem a listem vždy pouze jeden uzel skupiny, ale je možno použít rozvětvenější strom skládáním skupin.

Vykreslení HTML kódu lze provést nezávisle na celistvosti stromu, nahrání hodnot a validace je ale závislá na přítomnosti kořene. Třída Form poskytuje mezistupeň mezi HTTP požadavkem a jednotlivými položkami. V případě potřeby ji lze napojit na jiný zdroj vstupních dat, což lze využít při automatizovaném testování formulářů.

Vlastní třídy programátor většinou odvozuje od třídy Form, zřídka od třídy FormGroup. Pro běžné formuláře jsou postačující třídy položek, které jsou součástí frameworku, programátor si ale může vytvořit vlastní třídu položky.

6.4.1 Třída FormGroup

Základní třída FormGroup slouží jako kolekce položek (nebo dalších skupin) a její metody lze rozdělit na dvě skupiny.



Obrázek 11: Třída FormGroup

První skupinou jsou metody Add(), Remove() a Find(), tj. pro manipulaci s kolekcí položek. Tyto operují nad chráněnou vlastností Items. Metoda Find() vyhledává položku podle jejího názvu a to i rekurzivně ve všech potencionálních podskupinách.

Druhou skupinou jsou metody pro hromadné zpracování pomocí rekurze. Metody jako Load() volají tutéž metodu na všechny prvky skupiny. Metody Render*() navíc při vykreslování na začátku a na konci vykreslí obal pomocí instance FormGroupLayout, která je uložena ve vlastnosti Layout. Framework poskytuje tři základní rozvržení pomocí HTML - fieldset, vertikální tabulku a horizontální tabulku.

6.4.2 Třída Form

Třída Form je odvozena od FormGroup, je tedy také kolekcí a pro práci s ní lze použít metody zděděné z třídy FormGroup. Rozdílné chování mají ovšem metody pro rekurzivní zpracování.

6.4.2.1 Metody Metoda Load() před vyvolání svých prvků nejdříve načte do vlastností SubmittedValues a SubmittedFiles kolekce z aktuálního požadavku. Následně podle přítomnosti hodnoty s klíčem dle vlastnosti SubmittedInputHiddenName nastaví vlastnost Submitted, tedy jestli jsou součástí požadavku vyplněná uživatelská data. Pro zefektivnění je metoda Load() vykonána pouze při prvním vyvolání a je uložen příznak do vlastnosti Loaded. Pro opětovné vynucení načtení dat je třeba zavolat metodu ForceLoad().

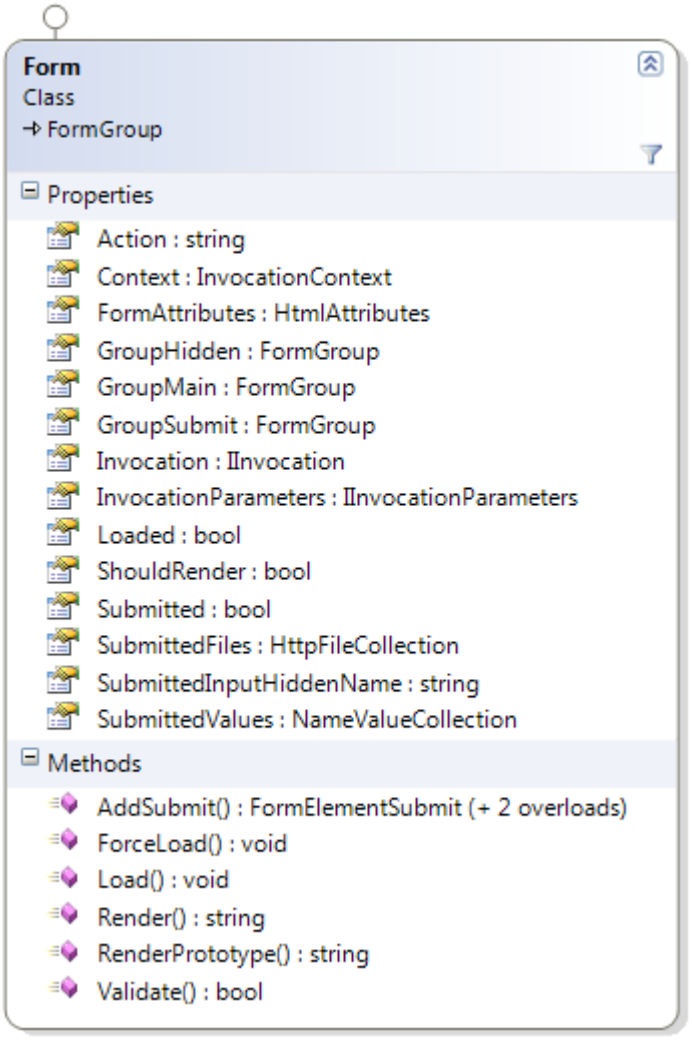
Metoda Validate() před provedením rekurze zavolá metodu Load() a svůj kód provede pouze tehdy, pokud je příznak odeslaní nastavena na true. V opačném případě Validate() vrací false, tedy (prázdná) obdržená data nejsou validní.

Metoda Render() podobně jako u FormGroup generuje kolem HTML svých prvků, ale nepoužívá FormGroupLayout. Místo toho generuje HTML element form, jehož atributy lze ovlivnit skrze vlastnost FormAttributes. Některé z atributů jsou dány vlastnostmi a jsou proto ignorovány. Jsou to atributy action, id, method a enctype.

Metoda AddSubmit() je pomocná a slouží ke snadnému doplnění odesílacího tlačítka do formuláře. To se přídává vždy do skupiny GroupSubmit, tj. na konec formuláře, a jeho text lze určit prvním parametrem. V případě vynechání je použit výchozí text.

6.4.2.2 Vlastnosti Vlastnosti GroupHidden, GroupMain a GroupSubmit jsou zkratky ke třem výchozím prvkům v kolekci Form. Všechny tři jsou typu FormGroup a jsou vykresleny v uvedeném pořadí. První skupina se vykresluje neviditelně pro uživatele, GroupMain je určena pro samotný obsah formuláře a GroupSubmit je určena pro odesílací tlačítko.

Vlastnost Action specifikuje hodnotu atributu action HTML elementu form. Pokud není vyplněno, použijí se vlastnosti Invocation a InvocationParameters k vygenerování odkazu. Pokud ani ty nejsou vyplněny, použije se prázdná hodnota, tj. formulář se pošle na stejnou URL, jako na které byl zobrazen.



Obrázek 12: Třída Form

Vlastnost ShouldRender je pouze pomocná a není třídou přímo použita. Programátor ji může využít k uložení informace, že se nemá formulář již vykreslovat do HTML výstupu, například pokud byl výsledek metody Validate() pravda a data formuláře byla tedy zpracována.

6.4.3 Položky formuláře

Třída FormElementBase obsahuje řadu vlastností, které slouží pro úpravu jejího vykreslení. To se vždy provádí v několika krocích zanoření.

Základní metoda Render() volá metodu RenderPart() postupně s parametry ContainerBegin, Main, Additional a ContainerEnd, přičemž předá řízení buď zpětnému volání definovanému ve vlastnosti FormRenderers, pokud příslušný klíč existuje, nebo použije výchozí vykreslení pomocí metody RenderPrototype(). Obě varianty následně mohou volat opět RenderPart() s jinou částí položky k vykreslení (například Main obsahuje Element a Label).

Všechny vlastnosti s názvem *Attributes definují atributy obaly HTML elementů vykreslených v příslušných volánch RenderPrototype().

Položka s nastavenou vlastností Frozen bude vynechávat provádění metody Load(). Vlastnosti Disabled a ReadOnly informují kód pro vykreslování HTML elementu, že mají přidat atribut disabled, resp. readonly. Vlastnost View značí, že by se editační element neměl vykreslit, ale měla by se pouze zobrazit hodnota.

Položce lze nastavit nadpis pomocí vlastnosti Label. Tu lze vyplnit také parametrem konstruktoru. Dále lze k položce doplnit delší vysvětlující texty přidáním do kolekce Notes a pomocně klientské skripty přidáním do kolekce Javascripts.

Výchozí hodnota položky se nastavuje ve vlastnosti DefaultValue. Po provedení metody Load() je do vlastnosti Value přiřazena hodnota načtená od uživatele, pokud byla nějaká obdržena (formulář byl poslán). Následně se do RenderValue přiřadí Value, popř. DefaultValue, pokud je Value nepřiřazena.

Metoda Validate() nejprve aplikuje všechny filtry z kolekce Filters a poté spustí všechny validátory z kolekce Validators. Výsledek validace je dán konjukcí výsledků jednotlivých validátorů.

6.4.3.1 FormElementBoolean Slouží ke vstupu hodnoty pravda/nepravda. Typ vstupu lze ovlivnit nastavením vlastnosti Mechanism, výchozí chování je použití zatržítka.

6.4.3.2 FormElementFile Slouží k nahrání souboru na server, který je následně uložen do adresáře InvocationApplication.FilesPath. K jeho přesunutí lze použít metodu MoveFile().

6.4.3.3 FormElementFileText Slouží k výběru souboru z těch, které jsou již na serveru nahrané a umožnuje také manipulaci s nimi pomoci správce souborů. Místo základního správce, který je součástí framework lze integrovat některou z komerčních komponent jako ckfinder nebo FileVista.

6.4.3.4 FormElementHidden Slouží k doplnění HTML pole input typu hidden. Hodnota je typu string.

6.4.3.5 FormElementHtml Slouží k doplnění HTML kódu do formuláře. Nečte žádnou hodnotu.

6.4.3.6 FormElementPassword Slouží k doplnění HTML pole input typu password. Hodnota je typu string. Jde o variaci FormElementText.

6.4.3.7 FormElementReset Vykreslí tlačítko Reset, jehož činnost řeší prohlížeč.

6.4.3.8 FormElementSelect Řeší výběr jedné položky z nabídky. Výchozí mechanismus je HTML element select. Jde o generickou třídu, lze tedy vrátit položku libovolného typu převeditelného na string (tedy například i int nebo enum).

6.4.3.9 FormElementSubmit Vykresluje odesílací tlačítko.

6.4.3.10 FormElementText Slouží k doplnění HTML pole input typu text. Hodnota je typu string. Jde o speciální případ generické třídy FormElementInput, která umožňuje určit, do jakého typu se má obdržený řetězec přeložit.

6.4.3.11 FormElementTextArea Slouží ke vstupu delšího textu. U tohoto elementu lze zvolit, zda má být vstupem prostý text (editor PlainText), nebo má být použit WYSIWYG prvek (editor Html, použita je open-source komponenta TinyMCE).

6.4.4 Vytvoření vlastní položky

Pro vytvoření vlastní položky je vhodné použít generickou abstraktní třídu FormElement místo třídy FormElementBase, protože poskytuje jednodušší způsob a umožňuje určit typ hodnoty, která bude z formuláře získána.

Všechny třídy se starají o své vykreslení do HTML a proto musí implementovat metodu RenderPrototype(), která poskytne implementaci pro části ElementContent a ElementAdditional, které nejsou součástí FormElementBase.

Pokud se má položka nejen vykreslovat, ale také zpracovávat nějakou hodnotu poslanou uživatelem, je třeba doplnit implementaci pro metodu LoadValue().

Celý postup si lze nejsnadněji představit na komentovaném příkladu implementace třídy FormElementTextArea.

```
// odvozeni urcuje, ze hodnota polozky bude typu string
public class FormElementTextArea : FormElement<string>
{
    // toto je doplnujici vlastnost, ktera muze byt predana v˘konstruktoru
    public FormElementTextAreaEditor Editor { get; set; }
```

```
// toto je doplnujici vlastnost; vsechny doplnujici
//    vlastnosti, ktere nelze predat v~kostruktoru by mely mit
//    set i get
public bool ViewAsHtml { get; set; }

// rozsirujici konstruktor pro pohodlne pouziti
public FormElementTextArea(string id, string label, FormElementTextAreaEditor editor)
    : base(id, label)
{
    Editor = editor;
    ViewAsHtml = false;
}

// konstruktor se zakladni signaturou, neni povinny, ale doporuceny
public FormElementTextArea(string id, string label)
    : this(id, label, FormElementTextAreaEditor.Default)
{
}

// implementace je treba, pokud ma polozka cist hodnoty
// z~uzivatelskeho vstupu
protected override void LoadValue()
{
    // vlastnost Form vrati koren stromu formulare
    // vlastnost Name je vetsinou stejna jako Id a je to klic,
    //    pod kterym budou data v~kolekci SubmittedValues
    Value = Form.SubmittedValues.Get(Name);
}

public override string RenderPrototype(FormRenderParts parts)
{
    // je treba napsat implementaci pro ElementContent a ElementAdditional
    // implementace pro ostatni casti lze prepsat, ale ve vetsine
    // pripadu to neni vhodne
    if (parts == FormRenderParts.ElementContent)
    {
        // vzdy je treba osetrit oba stavy vlastnosti View
        if (View)
        {
            // FormElementTextArea je napsan i pro editaci pomoci WYSIWYG
            // implementace pro View == true byva obvykle kratsi
            if (ViewAsHtml)
                return RenderValue;
            else
            {
                string value = RenderValue;
                if (Editor == FormElementTextAreaEditor.Html)
                    value = value.Replace("<br␣/>", "\n").Replace("</p>", "\n").Replace("</
                        div>", "\n");
                value = Utilities.StripTags(value);
                return HttpUtility.HtmlEncode(value).Replace("\n", "<br␣/>\n");
            }
        }
        else
```

```
            {
                // delsi byla kod pro vystup editacniho pole
                StringBuilder o~= new StringBuilder();
                // autor nove tridy by mel zohlednit vlastnost ElementAttributes
                var attributes = ElementAttributes.Clone() as HtmlAttributes;
                attributes["id"] = Id;
                attributes["name"] = Name;
                if (! attributes.ContainsKey("cols"))
                    attributes["cols"] = "40";
                if (! attributes.ContainsKey("rows"))
                    attributes["rows"] = "5";
                string classAttribute = "PlainText";
                if (Editor == FormElementTextAreaEditor.TinyMCE)
                    classAttribute = "Html␣TinyMCE";
                o.Append("<textarea");
                o.Append(attributes.ToString(classAttribute, null));
                o.Append(">");
                o.Append(HttpUtility.HtmlEncode(RenderValue));
                o.Append("</textarea>");
                if (Editor == FormElementTextAreaEditor.TinyMCE)
                {
                    if \footnote{Form != null) && (Form.Context != null}
                        Form.Context.Flags.Add(HtmlViewInvocationContextFlags.TinyMCE);
                }
                return o.ToString();
            }
        }
        // tato cas je povinna, i kdyz vraci jen prazdny retezec
        else if (parts == FormRenderParts.ElementAdditional)
        {
            return string.Empty;
        }
        else // pro vsechny ostatni je treba zavolat rodicovskou tridu
            return base.RenderPrototype(parts);
    }
}
```

6.5 Příklady použití

Jak bylo uvedeno v předchozí sekci, účelem implementovaný vlastností je poskytnout programátorovi několik možných postupů s různou úrovní složitosti a oddělení dat, aby si mohl pro svůj konkrétní případ zvolit takové řešení, které nejvíce odpovídá množství času, který má na implementaci daného formuláře vyhrazeno, a přitom výsledek vždy vyhovoval základním požadavkům na použitenost.

6.5.1 Bez vlastní třídy

Tato úroveň řešení je velmi přímočará. Formulář je nejdříve naplněn existujícími elementy, validován, zpracován a vykreslen. Je ideální pro unikátní, jednorázové formuláře, především pokud nejsou spojeny s business entitami. Jeho využití zároveň velmi

výhodné pro prototypovací fázi vývoje, protože výsledný formulář může v kombinaci s předpřipraveným obecným kaskádovým stylem poskytnout i nezkušenému uživatel (tedy například zadavateli projektu) dostatečnou představu o budoucím fungování formuláře.

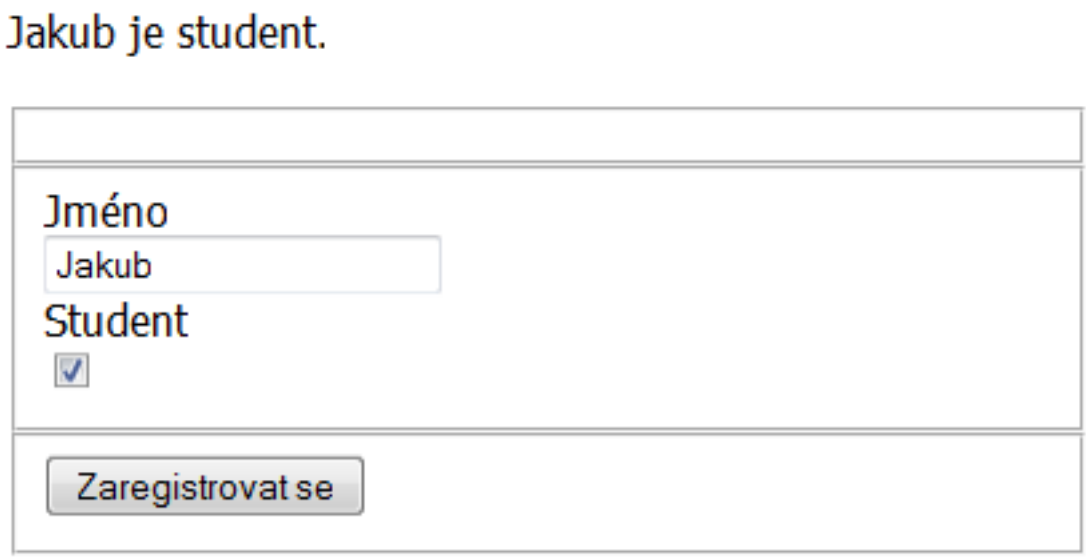
```
public class Test1 : TextClassTemplate
{
    public override void Render()
    {
        Form form = new Form("form1");

        var e1 = form.GroupMain.Add(new FormElementText("jmeno", "Jméno"));
        e1.Validators.Add(new FormValidatorRequired());
        var e2 = form.GroupMain.Add(new FormElementBoolean("student", "Student"));

        form.AddSubmit("Zaregistrovat se");

        if (form.Validate())
        {
            __w.Write("{0} {1} student.<br /><br />",
                e1.Value,
                (form.Find<FormElementBoolean>("student").Value ? "je" : "není")
            );
        }

        if (form.ShouldRender)
            __w.Write(form.Render());
    }
}
```

6.5.2 Nezávislé vykreslení

Tento případ je netradiční tím, že z hlediska vykreslení formuláře do HTML představuje protipól k předchozímu případu. V praxi se často stává, že z grafického návrhu vzejde malý a velmi úzce zaměřený formulář, jehož vzhled nelze pomocí výchozího vykreslení dostat pouze pomocí kaskádových stylů. Typickým příkladem je právě ukázaná registrace do newsletteru, která bývá vložena na každou stránku prezentace.

Pro tyto případy lze tedy zvolit kombinaci předpřipraveného generovaného HTML. V tomto příkladu je do jinak statického HTML vloženo zobrazování chybových hlášení pro validátor.

```
public class Test2 : TextClassTemplate
{
    public override void Render()
    {
        Form form = new Form("form2");

        form.SubmittedInputHiddenName = "email";
        var e1 = form.GroupMain.Add(new FormElementText("email", "E-mail"));
        e1.Validators.Add(new FormValidatorEmail());
```



Obrázek 13: Výstup formuláře bez použití vlastní třídy (po odeslání)

```
        if (form.Validate())
        {
            _w.Write("{0} se zaregistroval.<br /><br />", e1.Value);
            form.ShouldRender = false;
        }

        if (form.ShouldRender)
            _w.Write(Template.Create("Form2").Render().Text);
    }
}
```

```
<viewdata Form="Form form" />
<form action="" method="post">
    <div>
        <h2>Registrace newsletteru</h2>
        <label for="email">E-mail:</label> <input type="text" name="email" id="email" />
        ${form.GroupMain["email"].RenderPart(FormRenderParts.Errors)}
        <input type="submit" value="Registrovat" />
    </div>
</form>
```

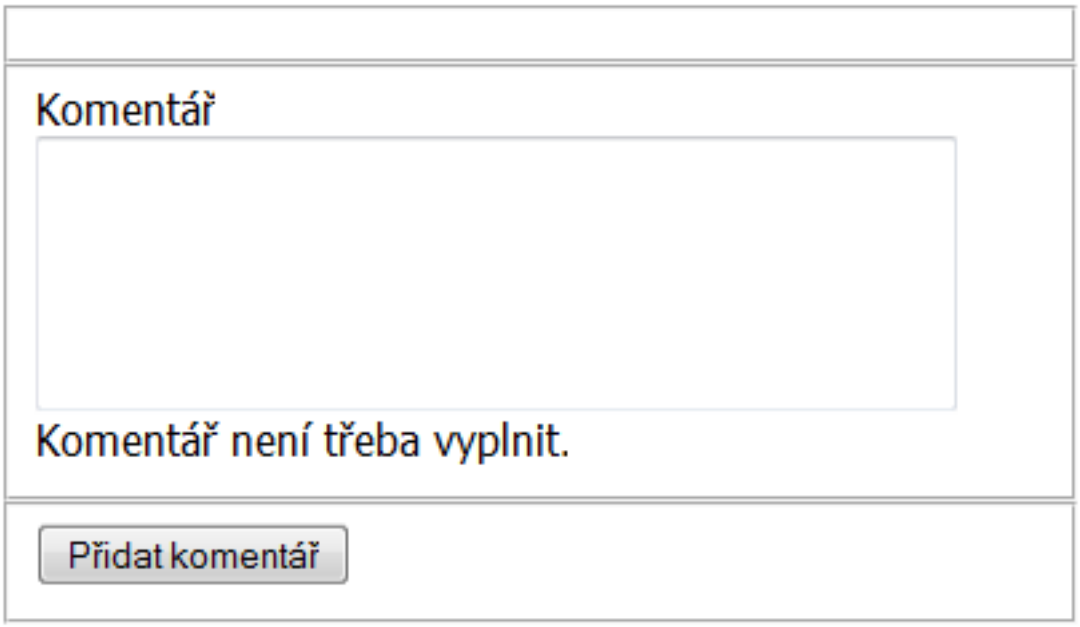
6.5.3 S vlastní třídou

Vyšším stupněm při implementování formuláře je použití stejného přístupu jako Webforms. V tomto příkladu vidíme, že vytváření struktury formuláře je vloženo do samostatné třídy a při zpracování dat se přistupuje přímo k jednotlivým elementům.

Obrázek 14: Výstup formuláře vykresleného nezávisle

Výhoda tohoto postupu spočívá v možnosti oddělení libovolné části práce s formulářem do samostatného bloku kódu a v možnosti přetížení výchozího chování, například metody Validate(). To může být užitečné zejména tehdy, pokud chceme využít implementovaný generátor HTML, ale potřebujeme upravit větší množství parametrů vzhledu.

```
class Form3 : Form
{
    public FormElementTextArea Komentar;

    public Form3()
        : base()
    {
        Komentar = new FormElementTextArea("Komentar", "Komentář");
        Komentar.Notes.Add("Komentář není třeba vyplnit.");
        GroupMain.Add(Komentar);

        AddSubmit("Přidat komentář");
    }
}

public class Test3 : TextClassTemplate
{
    public override void Render()
    {
        Form3 form = new Form3();

        if (form.Validate())
            w.Write(form.Komentar.Value + "<br /><br />");

        if (form.ShouldRender)
            w.Write(form.Render());
    }
}
```



Obrázek 15: Výstup formuláře implementovaného vlastní třídou

6.5.4 S oddělením dat

Úpravou třídy z předchozího příkladu můžeme dosáhnout oddělení dat, podobně jako je tomu v MVC frameworcích. Všechny komponenty formuláře změníme na přístupovou úroveň protected a do třídy formuláře přidáme vnitřní třídu a jako její instanci veřejný člen reprezentující data obdržená od uživatele. Napojením na událost OnLoaded můžeme takto vzniklý model vyplnit a následně použít v kódu zpracovávajícím data.

```
class Form4 : Form
{
    protected FormElementTextArea Komentar;

    public class FormData
    {
        public string Komentar;
    }

    public FormData Data;

    public Form4()
        : base()
    {
        OnLoaded += new FormEventDelegate(Form4_OnLoaded);
```

```
        GroupMain.Label = "Přidání nového komentáře";

        Komentar = new FormElementTextArea("Komentar", "Komentář");
        GroupMain.Add(Komentar);

        AddSubmit();
    }

    void Form4_OnLoaded(FormElementBase element, FormEventType eventType)
    {
        Data = new FormData();
        Data.Komentar = Komentar.Value;
    }
}

public class Test4 : TextClassTemplate
{
    public override void Render()
    {
        Form4 form = new Form4();

        if (form.Validate())
        {
            __w.Write(form.Data.Komentar + "<br /><br />");
        }

        if (form.ShouldRender)
            __w.Write(form.Render());
    }
}
```

Toto řešení můžeme dále vylepšit doplněním další úrovně abstrakce pomocí rozhraní, případně pomocí návrhového vzoru Inversion of Control [9]. Tím lze implementaci formuláře naprosto oddělit od zpracování dat.

```
interface IMyFormData
{
    public string Komentar { get; }
}

interface IMyForm : IForm
{
    public IMyFormData Data { get; }
}

public class Test4 : TextClassTemplate
{
    public override void Render()
    {
        IMyForm form = IocContainer.Create<IMyForm>();

        if (form.Validate())
        {
            __w.Write(form.Data.Komentar + "<br /><br />");
```

```
            }

            if (form.ShouldRender)
                __w.Write(form.Render());
        }
    }
```

7 Persistence dat a O/RM

Persistence dat, neboli jejich ukládání do trvalé paměti (většinou databáze uložené na pevném disku), je prakticky nejdůležitější součástí webové aplikace a našem případě i webové prezentace. Pokud by projekt nevyžadoval persistenci dat nebo bychom neočekávali možnost, že ji bude v budoucnu vyžadovat, zvolíme jistě k jeho výrobě takové prostředí, které tomu více uzpůsobeno.

Dosavadní praxe mi ale poskytla důležitou informaci týkající se použití persistence dat a to, že dostatečně velký podíl webových prezentací ze zadání nebo později vyžaduje nějakou formu persistence dat, aby se vyplatilo mít připraveno řešení, které je těsně integrováno v rámci celého frameworku.

7.1 Základní pojmy

Protože cílem frameworku je co nejvíce usnadnit tvorbu takových webových aplikací, z jejichž povahy není třeba samostatný návrh databáze, vychází při řešení struktury databáze z objektového doménového modelu. Tento je reprezentací dat nezávisle na jejich implementaci a používá zvláštní pojmenování pro jednotlivé prvky této reprezentace.

Entitou [10], někdy také business entitou nebo doménovou entitou, je myšlena kolekce informací, kterou lze samostatně vyčlenit. Entitou může být například prodávaný produkt, článek, nebo uživatel. Meta informace o struktuře podobných entit se nazývá entitní typ. Jednotlivá informace v rámci entity se označuje jako atribut entity. Entita může být také spojena vztahem s jinou entitou. Entita může zahrnovat také operace pracující s jejími atributy a vztahy.

Tento abstraktní koncept se následně mapují do aplikační vrstvy a databázové vrstvy. V rámci tohoto frameworku na jedná straně stát jazyk C# a na druhé relační databáze.

V aplikační vrstvě je entitní typ reprezentován třídou, entity jejími instancemi, atributy a vztahy vlastnostmi s příslušným typem a operace jsou metody těchto tříd.

V databázové vrstvě je entitní typ reprezentován tabulkou, entity jsou řádky tabulky, atributy a vztahy sloupce tabulky. Operace bývají implementovány jako uložené procedury.

7.2 Přehled dostupných řešení

Existuje řada různorodých řešení, ale v praktickém vývoji webových prezentaci jednoznačně nejčastěji použita relační databáze. Z toho důvodu bylo její použití zvoleno i pro tento framework, a protože je .NET objektové prostředí, je zřejmé, že ideálním řešením je objektově-relační mapování (O/RM, častěji pouze ORM).

Základní požadavky na použitou technologii opět vycházejí ze specializace frameworku:

1. Z dlouhodobého hlediska musí mít řešení dostatečné zázemí a údržbu ze strany výrobce.

2. Nesmí představovat dodatečné finanční náklady.

3. Musí z databázových strojů podporovat alespoň mySQL a Microsoft SQL Server.
4. Rozšířitelnost entit pomocí třídní hierarchie.
5. Struktura dat musí vycházet z objektového modelu, nikoliv z databáze. Framework musí mít možnost zjistit, které pole entity jsou persistentní.

Body 4. a 5. je nejlepší demostrovat na praktickém příkladu. Jedním z bodů integrace je definice společného primárního klíče jako celočíselnou vlastnost EntityId. Tím zajistíme, že každá entita je jednoznačně identifikovatelná kombinací dvou hodnot - svého typu a celočíselného klíče. Proto je třeba umožnit následující konstrukci.

```
public class Entity
{
    public int EntityId { get; set; }

    // dalsi operace
}

public class UserBase : Entity
{
    public string Login { get; set; }

    // dalsi operace
}

public class User : UserBase
{
    public bool IsManager { get; set; }
}
```

Implementace třídy Entity a UserBase se nacházejí ve frameworku a ten zná pouze tyto dvě. Ve skutečnosti ale bude databázové struktuře odpovídat třída User, která se nachází přímo v projektu a rozšiřuje entitu uživatele o novou vlastnost IsManager. V kódu frameworku však tuto skutečnou třídu nelze použít a tento s ní bude pracovat pouze přes rozhraní poskytované třídami Entity a UserBase.

Důsledkem toho je, že pokud řešení neposkytuje možnost pracovat s persistetními entitami jako s jejich nadřazenými třídami, je takové řešení pro účely tohoto frameworku nevhodné.

7.2.1 Vlastní O/RM

První řešení je samozřejmě implementace vlastní knihovny O/RM. To ovšem vyžaduje další prostředky věnované na údržbu frameworku a ve srovnání s ostatními možnostmi by mělo řešení horší vlastnosti. Proto je dlouhodobého hlediska nevhodné.

7.2.2 LINQ to SQL

LINQ to SQL je zástupce ze skupiny tzv. generátorů. Tyto nástroje vytvářejí zdrojový kód na základě struktury databáze. Toto řešení tedy není vhodné, protože neumožňuje potřebným způsobem řešit dědičnost.

Další, související, nevýhodou je právě dotazovací jazyk LINQ, který je sice výrazně jednodušší na zápis než tvoření dotazu pomocí speciálních funkcí, ale při jeho použití musí programátor mít k dispozici přímo persistetní třídu a zápis nelze dostatečně ošetřit následnou úpravou výrazu při překladu na SQL.

7.2.3 Subsonic

O/RM Subsonic poskytuje mnoho zajímavých vlastností, mezi jinými i možnost volby mezi použitím návrhového vzoru Active Record, nebo přístupu označovaného jako POCO (Plain Old C# Object, označení vzniklé z dřívějšího Plain Old Java Object), kde entity nejsou nijak přímo spojeny s vrstvou persistence.

Jeho nevýhodou je ovšem, že se jedná o menší řešení, vhodnější pro osobní použití a nelze s jistotou očekávat opravy chyb. Bez potřebné podpory ze strany autora by toto řešení mohlo způsobit problémy v případě potřeby použít nějakou pokročilejší vlastnost, která v něm není dosud dostatečně implementována.

7.2.4 NHibernate

NHibernate (http://nhforge.org/) je reimplementací rozšířeného Java ORM Hibernate pro platformu .NET. Jedná se o open-source projekt, který je momentálně vyvíjen komunitně a může čerpat z dlouholetých zkušenosti z velmi blízkého prostředí jazyka Java. Z nekomerčních ORM má jednoznačně nejširší podporu a je zaručen jeho další rozvoj. Díky tomu se jedná o téměř ideální volbu.

7.2.4.1 Castle.ActiveRecord Jednou z mála nevýhod knihovny NHibernate je, že z Hibernate dosud převzal jen konfiguraci entit jen za pomocí XML souboru. Zatímco pro Hibernate existuje řešení v podobě přídavného projektu Annotations, vyvíjeného společně s jádrovým projektem, tak projekt NHibernate sám takovou funkcionalitou nemá.

Existují ovšem dva nezávislé projekty, které umožňují snadnější konfiguraci, každý jiným způsobem. FluentHNibernate (http://fluentnhibernate.org/) používá speciální třídu a lambda výrazy a Castle.ActiveRecord (http://www.castleproject.org/activerecord/) poskytuje konfiguraci podobně jako Annotations, tedy pomocí .NET atributů. Zároveň ale k NHibernate přidává velmi snadnou použitelnou vrstvu založenou na návrhovém vzoru Active Record.

```
[ActiveRecord]
public class Pracovnik : ActiveRecordBase<Pracovnik>
{
    [PrimaryKey]
    public int Id { get; set; }
```

```
    [Property]
    public string Jmeno { get; set; }

    [BelongsTo]
    public Pracovnik Nadrizeny { get; set; }
}

var jakub = Pracovnik.FindFirst(Expression.Eq("Jmeno", "Jakub"));
if (jakub.Nadrizeny != null)
    w.Write(jakub.Nadrizeny.Jmeno);
```

Kombinace NHibernate a Castle.ActiveRecord tedy poskytuje snadnou konfiguraci a zároveň prostor pro řešení komplikovanějších scénářů pomocí samotného NHibernate. Proto byla tato kombinace zvolena jako základ persistence pro tento framework.

7.3 Návrh a implementace

Cílem je zachovat co nejvíce činností pouze v rámci NHibernate a Castle.ActiveRecord a pouze definovat omezení a rozšíření pro snadnější integraci se zbytkem frameworku:

1. Sjednotit primární klíče všech entitních typů.
2. Vytvořit řešení pro lokalizaci entity.
3. Vytvořit vlastní strukturu meta-informací o entitních typech.

7.3.1 Sjednocení primárního klíče

Primární klíč lze řešit dvěma základními způsoby a to generování na straně aplikace, nebo generováním na straně databáze.

7.3.1.1 Generování na straně databáze Generování na straně databáze je v převážné většině případů realizováno pomocí celočíselných sekvencí. Výhoda tohoto přístupu je v efektivitě uložení, indexace a vyhledávání. Celočíselné atributy zabírají diskový prostor nejefektivěji a také je u nich nejrychlejší řazení, na kterém závisí výkon.

7.3.1.2 Generování na straně aplikace Při generování na straně aplikace je třeba zajistit, že vygenerovaný klíč bude unikátní (nebo alespoň dosáhnout dostatečně nízké pravděpodobnosti duplicity). Proto se nějčastěji používají UUID [11] (Universally Unique Identifier). Jejich generování zaručuje dostatečnou unikátnost, aby bylo možné později spojit dvě databáze bez kolize klíčů. Díky generování na straně aplikace lze také připravit celou sadu propojených záznamů bez potřeby rezervování klíče a komunikace s databází.

Nevýhodou je, že jen málo databázových strojů podporují tento typ nativně jako 128-bitové číslo, ale jen jako textový řetězec. Tím dochází ke zvětšení místa potřebného k uložení klíče a ztrátě výkonu.

Kompromisním řešením by mohlo být generování vlastního 64-bitového celočíselného klíče na straně aplikace. Na 64-bitových databázových strojích by měl poskytnout stejný výkon jako použití 32-bitového klíče, ale také výhody z toho, že je klíč záznamu znám ještě před jeho uložením.

7.3.1.3 Zvolené řešení Protože je framework specializován na webové prezentace, byla zvolena varianta generování 32-bitového celočísleného klíče na straně databáze. Pro tento typ webové aplikace jde o řešení dostačující a díky schopnostem NHibernate lze volbu relativně snadno změnit refaktorováním kódu.

7.3.2 Lokalizace entit

Lokalizací entit je myšlen takový koncept, který umožní mít pro jednu entitu texty přeložené pro každý z jazyků aplikace zvlášť. Například tisková zpráva bude mít pro všechny jazyky společné datum vydání a fotografii, ale pro každý jazyk bude zvlášť uložen text zprávy a její PDF příloha.

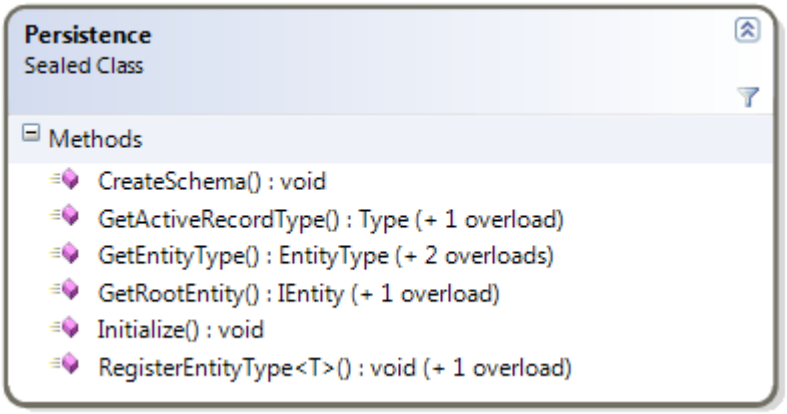
K lokalizaci entit lze přistoupit několika způsoby, přičemž zdánlivě mají všechny tyto možnosti pouze nevýhody. Je třeba si ovšem uvědomit, že tou nejdůležitější výhodou společnou pro všechny je samotný efekt lokalizace entit. Pro webové prezentace jde o velmi důležitou část řešení a nelze tedy pouze přenechat řešení na konkrétní aplikaci. Zároveň je třeba uvažovat pouze taková řešení, která budou umožňovat práci přímo s vlastnostmi tříd, aby se co nejvíce zachovalo silné typování.

7.3.2.1 Použití připojené tabulky Čistým řešením je vytvoření dvou tabulek pro každou entitu a vazby mezi nimi. Jedna tabulka by představovala nelokalizovanou část typu entity a druhá část lokalizovanou. Toto řešení ovšem není momentálně v Castle.ActiveRecord dostatečně podporováno a i konfigurací NHibernate přes XML soubor by bylo natolik složité, že by byl porušen cíl jednoduchosti použití frameworku.

Proto jde o řešení, které nelze použít jako obecné. Rozhodně ale není jeho použití vyloučeno pro specifické případy konkrétních aplikací a programátor může využít všechny schopnosti knihovny NHibernate.

7.3.2.2 Spojené duplicitní záznamy Naivnějším řešením je vytvoření separátního záznamu pro každý dostupný jazyk a při změně provádět synchronizaci nelokalizovaných atributů.

Při analýze entitních typů vyskytujících se prakticky ve webových prezentacích bylo zjištěno, že pro žádný z dosavadních projektů firmy by toto řešení nebyl problém. Všechny entity s lokalizovanými atributy jsou vytvářeny a měněny pomocí společných základních funkcí v administraci a proto lze zmíněnou synchronizaci implementovat centrálně.



Obrázek 16: Třída Persistence, metody pro práci s meta-informacemi

7.3.2.3 Zvolené řešení Díky své jednoduchosti bylo po analýze a ověření proveditelnosti zvoleno řešení s ukládáním samostatných záznamů pro každý jazyk v rámci jedné tabulky.

7.3.3 Struktura meta-informací

Notace použitá pro objektově relační mapování pomocí Castle.ActiveRecord poskytuje meta-informace potřebné pro přenesení informací obsažených v entitě z a do databáze. Neposkytuje ale dostatek informací k přenesení těchto informací mezi aplikací a uživatelem. Proto framework implementuje vlastní vrstvu meta-informací pro automatizaci této činnosti.

Tyto metainformace jsou uložené v instancích tříd EntityType a EntityAttribute, popřípadě třídách od nich odvozených. Přístup k nim je možný pomocí metod statické třídy Persistence.

7.3.3.1 Třída Persistence

7.3.3.1.1 Metody pro práci s meta-informacemi Nastavení základních parametrů pro použití persistence dat společně s vrstvou pro meta-informace provádí metoda Initialize(). Tu je třeba zavolat před registrací entitních typů. Registraci entitních typů lze provést následně pomocí metody RegisterEntityType(). Je třeba tyto metody zavolat v inicializační fázi aplikace, specificky implementací metody InitializeApplicationEntities() v rámci Global.asax.

Registrace vytváří mapu mezi třídou, která skutečně představuje entitu a jejími nadřazenými třídami a rozhraními. Aby byla nadřazená třída nebo rozhraní přidáno do mapy, je třeba je označit atributem EntityInterface. Mapování musí být směrem od rozhraní jednoznačné a je úkolem programátora konkrétní aplikace, aby to zajistil.

Například existují čtyři úrovně pro realizace uživatele administrace. Nejvyšší úrovní je třída UserEntity, která bude skutečnou třídou entitního typu. Tato třída se nenachází ve frameworku a definuje ji programátor konkrétní aplikace odvozením od UserEntityBase, může tedy rozšiřit atributy této entity (které následně doplní do meta-informací). Třídu UserEntity framework nezná, ale s entitním typem uživatele může pracovat skrze nadřazenou třídu UserEntityBase nebo její rozhraní IUserEntity. Obě jsou označeny atributem EntityInterface a proto budou součástí mapy. Poslední úrovní je generická třída Entity. Ta nebude mít jednoznačně přiřazen pouze jeden entitní typ, a proto označena není a nebude ani součástí mapy.

Typy tříd a meta-informace z mapy lze získat pomocí metod GetActiveRecordEntityType() a GetEntityType(). Obě jako parametr přebírají třídu nebo rozhraní registrované ve zmíněné mapě. Například pro zavolání GetActiveRecordEntityType<IUserEntity>() je vrácena instance Type popisující třídu UserEntity.

Metoda GetRootEntity() je zkratkou pro GetEntityType().RootEntity a vrací instanci třídy entitního typu označenou jako kořenová entita. Tato instance poskytuje jiným způsobem přístup k metodám pro práci s daty. Její výhoda je ve vynucení typu při předání jako parametru metody (kořenová entita bude vždy implementovat rozhraní IEntity).

Po registraci všech entitních typů lze využít zabudovaného nástroje NHibernate pro vytváření struktury databáze. Metoda CreateSchema() je zkratkou pro vytvoření tabulek pro všechny zaregistrované entitní typy.

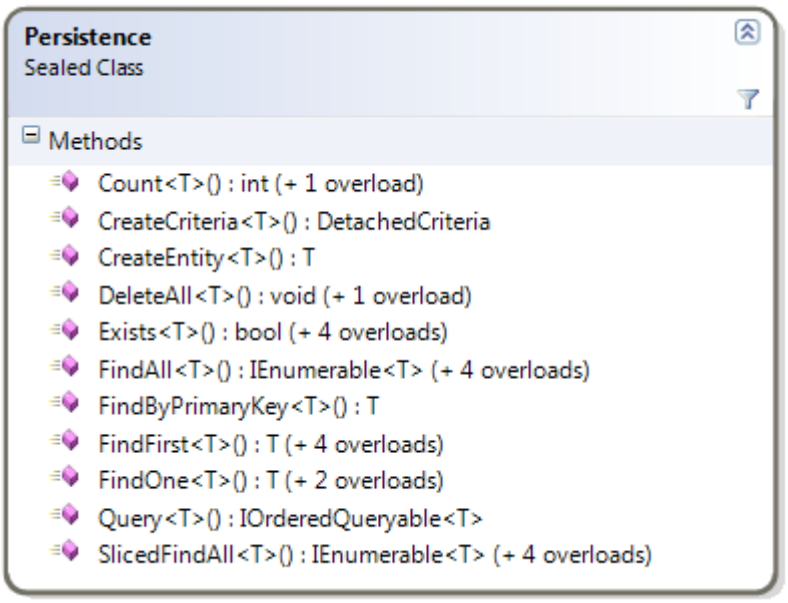
7.3.3.1.2 Metody pro práci s daty Skrze třídu Persistence jsou dostupné všechny metody pro práci s daty, které Castle.ActiveRecord poskytuje jako statické metody přiřazené u třídy entitního typu. Pokud ovšem nemáme jméno této třídy k dispozici, ale pouze implementované rozhraní, nelze tyto původní metody zavolat.

Z toho důvodu jsou reimplementovány (nejde tedy o použití reflexe) v třídě Persistence, přičemž jako generický parametr stačí předat třídu nebo rozhraní definované v mapě (viz. předchozí odstavce).

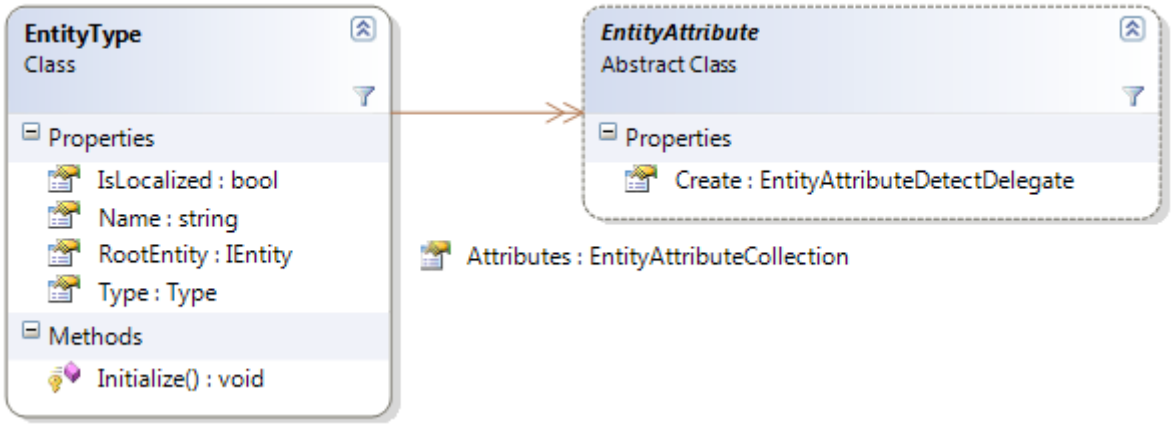
7.3.3.2 Třída EntityType Třída EntityType kromě vlastních meta-informací slouží dvěma dalším účelům. Udržuje v sobě kolekci instancí EntityAttribute a provádí detekci meta-informací o entitním typu jako celku.

Hlavní meta-informací je vlastnost IsLocalized, která určuje, jestli je třeba s entitami nakládat jako s vícejazyčnými (například synchronizaci dat při uložení). Entita je lokalizovaná právě tehdy, má-li alespoň jeden lokalizovaný atribut.

Inicializaci meta-informací pro konkrétní entitní typ je možné upravit odvozením vlastní třídy od EntityType a vrácení její instance při volání metody CreateEntityType() na kořenovou entitu. Tato metody by sama o sobě neměla provádět žádná nastavení.



Obrázek 17: Třída Persistence, metody pro práci s daty



Obrázek 18: Třída EntityType

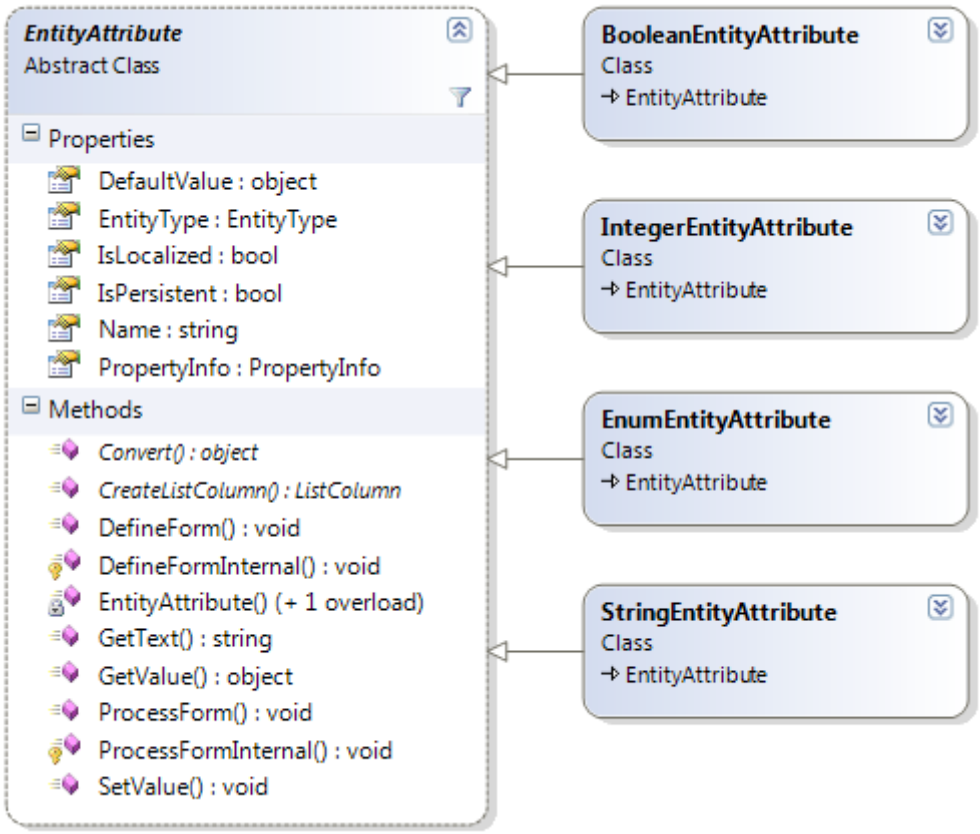
Při inicializaci má důležitou roli vlastnost EntityAttribute.Create. Jde o zpětné volání, které provádí základní detekci atributů podle jejich .NET typu v objektovém modelu. Pokud chce programátor zavést nový typ atributu v rámci celé aplikace, je výhodné doplnit toto volání o vlastní delegát.

7.3.3.3 Třída EntityAttribute Jak bylo již zmíněno, meta-informace slouží frameworku především pro automatizaci práce s entitami ve vztahu k uživatelskému vstupu. Každá třída odvozená od abstraktní třídy EntityAttribute definuje zvláštní chování, většinou pro určitý datový typ atributu entity. Toto chování je dáno metodami Convert(), CreateListColumn(), DefineFormInternal() a ProcessFormInternal().

Metoda Convert() má za úkol pokusit se co nejlépe převést informaci obdrženou jako parametr na skutečný typ entitního atributu. Například pokud metoda EntityAttributeBoolean.Create() obdrží jako parametr string „true“ nebo celé číslo 1, měla by vrátit bool hodnotu true.

Metoda CreateListColumn() je již podle svého názvu určena pro zobrazení sloupců ve výpisu entit v administraci. Obecně ale vytváří řetěz objektů, které jsou schopny převést hodnotu daného entitního atributu na informaci zobrazitelnou uživateli (bez ohledu na konkrétní rozvržení).

Metody DefineForm() a ProcessForm() jsou určeny k vytvoření formulářových polí, resp. k načtení obdržených hodnot z formuláře zpět do entity. Tyto metody by měl programátor volat, pokud pracuje s formulářem. Naopak metody DefineFormInternal() a ProcessFormInternal() by měl nahradit vlastní implementací, pokud definuje nový typ entitního atributu.



Obrázek 19: Třída EntityAttribute a některé typy atributů

Metody GetValue() a SetValue() jsou zkratky ke stejně pojmenovaným metodám instance PropertyInfo uložené v příslušné vlastnosti.

Třída EntityAttribute nese informace nejen o persistetních vlastnostech třídy entitního typu, ale o všech, u kterých dokáže detekovat vhodnou třídu odvozenou od EntityAttribute. Ke všem vlastnostem třídy, které jsou typu string, je vytvořena instance typu EntityAttributeString a podobně. To umožňuje oddělit skutečné entitní atributy od těch, které má k dispozici uživatel například přes administraci.

Programátor má možnost vynechat fázi autodetekce vhodného potomka EntityAttribute použitím .NET atributu EntityProperty u zvolené vlastnosti třídy entitního typu. Ten umožnuje:

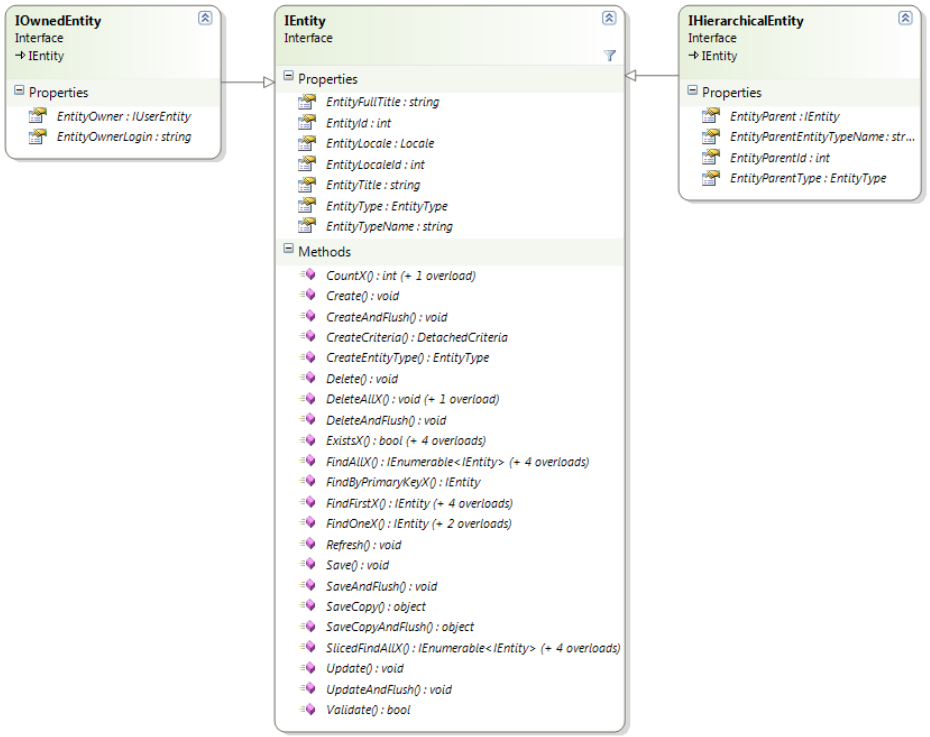
- Definovat konkrétní třídu meta-informace pomocí vlastnosti EntityAttributeType.
- Úplně vynechat zvolenou vlastnost z meta-informací nastavením IsEntityAttribute na false.
- Nastavit příznak lokalizace atributu entity IsLocalized.

7.3.4 Třída Entity

Generická třída Entity je základní rodičovskou třídou, kterou musí mít v hierarchii dědičnosti všechny třídy, u které programátor předává frameworku ke zpracování jako entity.

Sama o sobě obsahuje pouze malé množství aplikační logiky, ta je součástí tříd popsaných dříve v této kapitole. Třída Entity slouží především připravená implementace rozhraní IEntity. Toto rozhraní popisuje omezení a požadavky na vlastnosti a metody třídy entitního typu, aby mohla být korektně integrována do zbytku frameworku.

Tyto předpoklady a omezení si můžeme přehledně shrnout diagramem rozhraní IEntity a nepovinných rozhraní IOwnedEntity a IHierarchicalEntity. Implementací těchto rozhraní programátor na třídě entitního typu aktivuje doplňující omezení, která lze následně použít především pro účely co nejpřirozenějšího zobrazení těchto entit v administračním rozhraní.



Obrázek 20: Třídní diagram rozhraní IEntity

8 Cache

Jak uvádí anglická Wikipedie [12], cache (česky také vyrovnávací paměť) je součástí programu, jejímž cílem je zvýšit výkon programu ukládáním dat do mezipaměti, aby se nemusela při každém požadavku získávat z původního zdroje. Jako teoretický koncept je tedy cache velmi užitečná a v podstatě všechny webové aplikace či prezentace by jejím použitím měly získat.

Cache je tedy z vnějšího pohledu kolekce, jejíž prvky nejsou uloženy až do ručního odstranění, ale samovolně mizí po dosažení platnosti určité podmínky, v převážné většině určitého času.

Ze své zkušenosti v oblasti vývoji webových aplikací vím, že v praxi se cache používá zřídka. Některé obtíže sice řeší volba vhodné implementace cache, popřípadě obměna mechanismu ukládání dat, ale hlavní problém zůstává bez ohledu na konkrétní technické řešení. Tímto problémem je určení, pro která data se má vyrovnávací paměť používat a kdy tato data uložená ve vyrovnávací paměti nejsou již platná a je třeba číst opět z původního zdroje. Definování těchto vztahů je často více časově náročné, než alternativní a méně efektivní formy optimalizace (jako například obejití některých vrstev abstrakce), a zejména u menších projektů se možnost využití vyrovnávací paměti při návrhu vůbec nebere v úvahu. Než tedy programátor začne tuto problematiku řešit, musí zvážit, jestli u jeho konkrétní aplikace je vůbec použití cache dostatečně přínosné, aby opravňovalo zkomplikování struktury.

8.1 Řešení ASP.NET, cache výstupu

Dejme si tedy předpoklad, že vyrovnávací paměť chceme nutně v naší aplikaci použít, a podívejme se na možnosti, které máme k dispozici. Základní ASP.NET a framework Webforms nám umožňují použití cache na třech tradičních úrovních: celé stránky, části stránky a dat.

První z těchto úrovní, cachování celých stránek, je teoreticky varianta s nejvyšším výkonem, ale v praxi se ukazuje, že jde také o variantu nejméně použitelnou. Na jedné straně spektra jsou aplikace, které jsou přiliš jednoduché a nízkonákladové, než aby se vyplatilo do realizace cache investovat dodatečný čas věnovaný návrhu a programování. Na druhé straně jsou pak aplikace, které na stránce obsahují alespoň jeden prvek, který z principu cachovat nelze, nebo by to bylo extrémně nepraktické. Typickým příkladem takovýchto aplikací jsou e-shopy, které na každé stránce obsahují dva takové prvky - současnou cenu zboží v košíku a jméno přihlášeného zákazníka.

Proto je mnohem častejší používání vyrovnávací paměti jen pro předem zvolené části stránek, resp. inverzní varianta, používání vyrovnávací paměti pro všechny kromě zvolených částí stránek (jako zmíněný obsah košíku, nebo informace o přihlášeném uživateli). Z mé zkušenosti se jedná o nejčasteji využívanou úroveň cachování, zejména díky její snadné implementaci. Každý fragment ve vyrovnávací má určený svůj unikátní název, složený z označení šablony, která jej generuje (například šablona detail produktu), a parametrů této šablony (například číslo produktu = 78).

Tento přístup s sebou ovšem přináší nepříjemnost, která je díky architektuře Webforms relativně dobře skryta, ale projeví se v MVC frameworcích. Jde totiž o cachování pouze na úrovni View, ovšem podle paradigmatu MVC se v této úrovni již nevyskytuje žádná aplikační logika a tedy všechna náročná práce je v okamžiku řešení provedena. Mnoho programátorů to ovšem neodradí a zejména webové aplikace v méně striktních jazycích, jak například PHP, tohoto řešení využívají, byť za cenu přesunutí části aplikační logiky do míst, kam nepatří.

8.1.1 ASP.NET MVC OutputCache

Experimentálně si lze ověřit, že OutputCache realizovaná ve frameworku ASP.NET MVC patří do první úrovně, tedy ukládá celou stránku. Stačí rozšířit ukázkovou aplikaci, která se generuje při založení nového projektu.

Po doplnění atributu OutputCache do k metodě Index() třídy HomeController zjistíme, že i po příhlášení pomocí odkazu v pravé horní části rozvržení bude na úvodní stránce opět tento odkaz, zatímco po vypnutí OutputCache bude vidět korektní odkaz k odhlášení.

```
[HandleError]
public class HomeController : Controller
{
    [OutputCache(Duration = 100, VaryByParam = "none")]
    public ActionResult Index()
    {
        ViewData["Message"] = "Welcome to ASP.NET MVC! " + DateTime.Now.
            ToLongTimeString();
        return View();
    }
}
```

8.2 Cache dat

Po této analýze jsem tedy usoudil, že cachování výstupu je pro udržovatelnost aplikace velmi nepraktické, a jeho použití je vhodné jen výjimečných případech. Navíc jak se ukáže později, lze cache výstupu bez velké ztrátě na výkonu nahradit cachováním dat.

Pojďme se tedy podívat, jak vypadá běžné použití cache dat v ASP.NETu, a také obecně v jiných prostředích či frameworcích.

```
// http :// msdn.microsoft.com/en-us/library/xhy3h9f9(v=VS.71).aspx
DataView Source = (DataView)Cache["MyData1"];
// 1. overeni
if (Source == null) {
    // 2. naplneni
    SqlConnection myConnection = new SqlConnection("server=localhost;Integrated Security=SSPI;
        database=pubs");
    SqlDataAdapter myCommand = new SqlDataAdapter("select * from Authors", myConnection);

    DataSet ds = new DataSet();
```

```
    myCommand.Fill(ds, "Authors");

    Source = new DataView(ds.Tables["Authors"]);
    Cache["MyData1"] = Source;
}
// 3. konzumace
MyDataGrid.DataSource=Source;
MyDataGrid.DataBind();
```

V tomto příkladu, převzatém z MSDN, je vidět nejčasteji používané schéma toku programu při použití vyrovnávacích pamětí libovolného druhu.

1. Dochází k ověření, zda se ve vyrovnávací paměti nacházejí platná data s identifikátorem, který požadujeme.
2. Pokud data nejsou k dispozici, tak jsou získány ze zdroje a uloženy do vyrovnávací paměti.
3. Nakonec jsou data použita.

Pokud tedy odhlédneme od konkrétní implementace a zjednodušíme, tak obdržíme následující kód.

```
object data;
if (CacheContains(KLIC))
    data = CacheRead(KLIC);
else
{
    // Create data
    data = 123;
    // End of create data
    CacheWrite(KLIC, data);
}
UseData(data);
```

K tomuto kódu můžeme také napsat alternativní, který vyrovnávací paměť nepoužívá.

```
object data;
// Create data
data = 123;
// End of create data
UseData(data);
```

Je naprosto jasné, že použití dat je společné pro obě varianty, a samozřejmě jej nelze vynechat. Můžeme ale kód s použitím cache upravit tak, aby se co nejvíce podobal tomu bez jejího použití? Jako první nás zřejmě napadne přesunout celý algoritmus získávání dat do samostatné funkce a to včetně řízení přístupu k vyrovnávací paměti. Oba upravené kódy by tedy vypadaly asi takto.

```
object GetData()
{
```

```
    object data;
    if (CacheContains(KLIC))
        data = CacheRead(KLIC);
    else
    {
        // Create data
        data = 123;
        // End of create data
        CacheWrite(KLIC, data);
    }
    return data;
}

object data = GetData();
UseData(data);
```

```
object GetData()
{
    object data;
    // Create data
    data = 123;
    // End of create data
    return data;
}

object data = GetData();
UseData(data);
```

Část kódu, která data používá, nyní již vypadá stejně a působí čistě a přehledně. Nyní je ještě potřeba nějak praktičtěji ošetřit práci s vyrovnávací pamětí - ideálně tak, abychom museli při doimplementování cache dopsat do nejméně kódu a zároveň zabránili nevhodnému přehmatu vlivem lidského faktoru a implementačního postupu copy-paste-edit. Úspěšně zde můžeme použít zpětných volání, která jsou v jazyku C# velmi dobře řešena. Skrze tyto úvahy pak získáváme výsledný prototyp optimálního kódu, podle kterého jsem řešil cache ve svém frameworku.

```
object GetData()
{
    object data;
    // Create data
    data = 123;
    // End of create data
    return data;
}

object data = CacheMagic(GetData);
UseData(data);
```

8.3 Implementace - jednoduchý příklad

Přejděme tedy nejdříve k příkladu, ve kterém si ukážeme základní použití. Představme si situaci, kdy je třeba všem návštěvníkům zobrazit v rohu stránky aktuální venkovní teplotu před jejich nejbližší pobočkou naší firmy (kupříkladu můžeme provozovat letní koupaliště). V takové situaci obvykle není žádoucí dotazovat se při každém požadavku na zobrazení stránky teploměru a také to není potřebné, protože teplota se s vysokou pravděpodobností nebude prudce měnit. Můžeme si ji tedy po nějakou dobu uchovat ve vyrovnávací paměti.

Předpokládejme, že pro jinou část webové aplikace již máme nadefinovaný enum se seznamem poboček, například takto:

```
enum Pobocka { Praha, Londyn, Madrid }
```

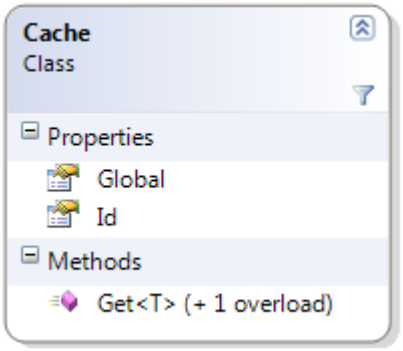
Vzhledem k tomu, že cachovací podsystém je schopen použít jako klíč libovolný typ, který lze převést na řetězec pomocí metody ToString(), můžeme tento enum poboček přímo využít. Vytvoříme tedy implementaci zdroje cachovaných dat, která bude převádět tento klíč na int, reprezentující okamžitou teplotu. Vytvoření této nové třídy je velmi jednoduché, stačí ji odvodit od generické třídy CacheItem s předáním typů klíče a výsledku (nebo jen typu výsledku, pokud není přídavný klíč potřebný). Následně do metody DetermineValue doplníme kód, který bychom za normálních okolností prováděli pro zjištění potřebné hodnoty. Tu nevracíme, ale přímo uložíme do vlastnosti CachedValue.

```
class VenkovniTeplota : CacheItem<Pobocka, int>
{
    public override void DetermineValue()
    {
        if (Key == Pobocka.Praha)
            CachedValue = DateTime.Now.Hour;
        else if (Key == Pobocka.Londyn)
            CachedValue = DateTime.Now.Hour - 3;
        else if (Key == Pobocka.Madrid)
            CachedValue = DateTime.Now.Minute;
    }
}
```

Nyní je třeba o tuto hodnotu na vhodném místě požádat. To lze provést jedním kombinovaným příkazem.

```
var teplotaVPraze = PoskiNET.Cache.Global.Get<VenkovniTeplota>(Pobocka.Praha).GetValue();
```

Každá instance třídy odvozené CacheItem reprezentuje jednu konkrétní variantu okolností definující možnou hodnotu. V tomto jednom příkaze vidíme všechny části, ze kterých se unikátní klíč skládá. První je identifikátor vyrovnávací paměti - ve většině případů není důvod, proč použít jinou než globální instanci. Druhou součástí klíče je celý název typu třídy odvozené od CacheItem. Třetí, nepovinnou, součástí je hodnota klíče předaného instanci. Pokud některou část unikátního klíče změníme (resp. použijeme jinou), obdržíme samostatnou instanci CacheItem.



Obrázek 21: Třídní diagram Cache

8.4 Implementace - podrobný pohled

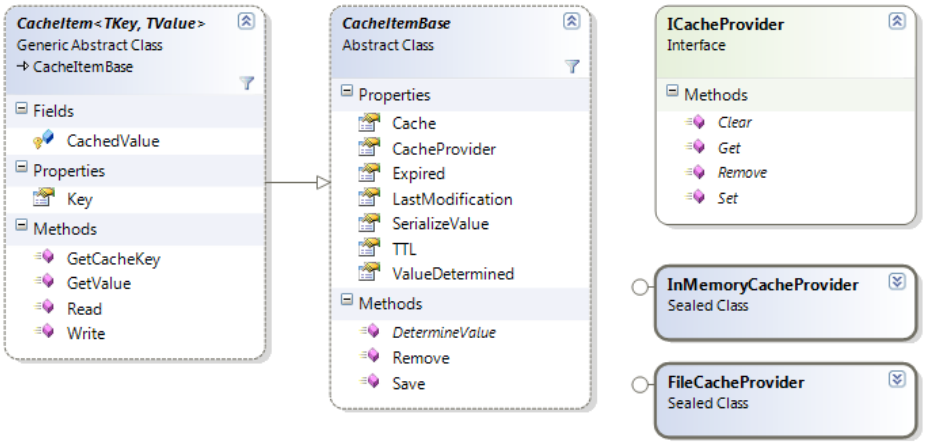
Konkrétní detaily a další možnosti, které nám poskytuje cachovací podsystém, nejlépe objasníme postupným průchodem třídními diagramy této části frameworku. Celá implementace je postavena na třech základních blocích:

1. třídě Cache, která představuje továrnu instancí tříd odvozených od CacheItem
2. abstraktní třídě CacheItem, která poskytuje součinnost při určování, ukládání a načítání samotných cachovaných dat
3. rozhraní ICacheProvider a jeho jednotlivých implementacích

8.4.1 Třída Cache

Z této trojice je snad nejméně zřetelná skutečná role třídy Cache. Jak je vidět z diagramu této třídy, obsahuje pouze jednu veřejně dostupnou metodu a to Get(). Tu jsme už potkali v předchozí sekci s příkladem a jejím úkolem je vytvořit instanci CacheItem a následně předat řízení příslušející implementaci ICacheProvider.

Tato třída není uzavřená a vývojář si od ní může případně odvodit vlastní implementaci s dodatečnými metodami pro pohodlnější práci s vlastními typy položek vyrovnávací paměti. Je třeba si ovšem uvědomit, že ačkoliv se navenek tato třída tváří jako kolekce, je její implementace bližší návrhovému vzoru továrna. Metoda Get() vytváří při každém vyvolání novou instanci CacheItem, protože sama si nemůže být vědoma již existujících položek. Tyto vytvořené vzory obdrží implementace ICacheProvider, do jejíž kompetence spadá rozhodnutí, zda do této instance pouze doplní hodnotu, nebo ji celou nahradí.



Obrázek 22: Třídní diagram CacheItem a ICacheProvider

8.4.2 Třída CacheItem

Jedinou činností, kterou je nutno udělat, abychom přidali další typ cachovaného údaje, je vytvořit novou třídu odvozením od generické třídy CacheItem.

Tato třída obsahuje již všechny potřebné části a je třeba pouze implementovat abstraktní metodu DetermineValue(). Jejím úkolem pouze vyplnit hodnotu vlastnosti CachedValue na základě klíče ve vlastnosti Key. Programátor nemůže při implementaci učinit žádný předpoklad, který sám nezajistí, mimo jiné se to týká také inicializovanosti InvocationApplication nebo toho, zda se zrovna vyřizuje požadavek.

Vlastnosti CacheProvider, TTL a Expired jsou virtuální a lze je nahradit vlastním kódem, přičemž platí stejné podmínky jako u DetermineValue(), tj. informace by měla být zjištěna pouze z klíče. CacheProvider musí vracet objekt implementující rozhraní ICacheProvider a jeho výchozí implementace vrací společnou výchozí instanci InMemoryCacheProvider.

Vlastnosti TTL a Expired jsou spojeny. TTL je zkratkou z anglického termínu Time to live a definuje dobu (v sekundách) od zjištění hodnoty, po kterou je tato hodnota platná. Vlastnost Expired porovnává aktuální čas s časem zjištění hodnoty (vlastnost LastModification) zvýšeným o TTL. Pokud je třeba změnit implementaci vlastnosti Expired, měla by tato nová implementace vždy vracet true, pokud původní implementace vrací true.

Z hlediska použití je nejdůležitější metoda GetValue(), která v případě, že ještě nebyla hodnota zjištěna nebo už expirovala (vlastnost Expired vrací true) spustí metodu DetermineValue() a uloží novou hodnotu do poskytovatele cache s klíčem určeným hodnotou výsledkem volání uzavřené metody GetCacheKey(). Metoda GetCacheKey() má za úkol převést hodnotu vlastnosti Key na typ string, přičemž pokud klíč implemen-

tuje rozhraní ICacheKey, použije se metoda GetCacheKey() klíče a v opačném případě se použíje metoda ToString().

8.4.3 Rozhraní ICacheProvider

Jak je vidět z předchozího třídního diagramu, je to konkrétní implementace CacheItem, která rozhoduje o tom, který z dostupných poskytovatelů úložného prostoru bude použit. To souvisí s odstíněním konzumace cachovaných dat. Konzumenta se totiž podrobnosti o způsobu uložení či aktuálnosti dat vůbec netýkají a je tudíž od nich naprosto odstíněn.

Při implementaci vlastního poskytovatele cache musí programátor implementovat čtyři základní metody Clear(), Set(), Remove() a Get(), přičemž se koncepcí jedná o speciální druh asociativní kolekce, kde klíč je typu string.

Důležitá vlastnost, kde se od kolekce liší je implementace metody Get(). Ta totiž může ošetřit nejen dostupnost prvku ve smyslu přítomnosti v kolekci, ale také může zkontrolovat, že platnost dat ještě nevypršela. Pokud vrátí null, je vynuceno vytvoření nové instance příslušného CacheItem.

8.5 TemplateCacheItem

Cachování výstupu šablon (myšleno instancí třídy Template) je ve frameworku z pohledu cache pouze jedním možným případem cachování dat. Těmito daty jsou instance třídy TemplateOutput a je pro ně připravena třída TemplateCacheItem speciálně pro tento účel odvozená od CacheItem. Samotná realizace se provádí doplněním zpětných volání do vlastností Template.ReadCache a Template.WriteCache. Jejich volání je provedeno těsně před a těsně za spuštěním instancí TemplateRenderer a vyžadují delegáty s následujíci signaturami:

```
public delegate void TemplateReadCacheDelegate(Template template, NameObjectCollection
    data, ref TemplateOutput output);
public delegate void TemplateWriteCacheDelegate(Template template, NameObjectCollection
    data, TemplateOutput output);
```

Jak je vidět, obě zpracovávající metody mají k dispozici instanci Template a kolekci předaných dat. Speciální ošetření instance TemplateOutput u čtení z cache umožňuje využít zřetězení delegátů. Jednotlivé metody přiřazené do zpětných volání se tedy mohou omezit na řešení pouze některých šablon, tj. například každý modul si může registrovat vlastní metody pro cachování šablon.

Následující kód ukazuje použití pro vytvoření běžné cache šablon. Proti tradičnímu přístupu je třeba mít na paměti, že bude vždy vrácena instance TemplateCacheItem, přičemž přítomnost hodnoty lze zjistit přes vlastnost ValueDetermined. Implementace TemplateCacheItem byla upravena tak, aby zabránila vzniku cyklu a hromadění instancí Template a NameObjectCollection. Jejím klíčem je tedy řetězec a její metoda GetValue() neprovádí výpočet.

Cache šablon tedy funguje spíše jako základní cache ASP.NET. Tu lze tedy samozřejmě použít místo TemplateCacheItem.

```
public void ReadTemplateCache(Template template, NameObjectCollection data, ref
    TemplateOutput output)
{
    if (template.Source == "Test")
    {
        string key = template.GetCacheKey(data);
        var cacheItem = PoskiNET.Cache.Template.Get<TemplateCacheItem>(key);
        if (cacheItem.ValueDetermined)
            output = cacheItem.GetValue();
    }
}

public void WriteTemplateCache(Template template, NameObjectCollection data, TemplateOutput
    output)
{
    if (template.Source == "Test")
    {
        string key = template.GetCacheKey(data);
        var cacheItem = PoskiNET.Cache.Template.Get<TemplateCacheItem>(key);
        cacheItem.SetAll(output, 60);
    }
}

public void DoSomething()
{
    Template.ReadCache += ReadTemplateCache;
    Template.WriteCache += WriteTemplateCache;
    Template t = new Template("Test");
    __w.Write(t.Render().Text);
}
```

9 Závěr

Úkolem této práce bylo odhalit a vyřešit problémy spojené s výrobou webových prezentací v ASP.NET. Mezi těmito problémy byla nejvýraznější témata týkající se vícejazyčnosti a ukládání dat.

Konečným cílem těchto řešení bylo vytvořit framework, který zvýší pružnost výroby webových prezentací a zároveň umožní pracovníkům specializovým na SEO nebo HTML/CSS podílet se bez asistence programátora větší měrou na výrobě projektu.

Tento cíl se podařilo splnit v plném rozsahu. Tímto byl učiněn velice důležitý krok pro rozvoj firemních aktivit v oblasti vývoje pro ASP.NET. Framework vytvořený v rámci této práce bude nasazen při výrobě komerčních projektů a bude dále rozvíjen na základě požadavků vznikajících z těchto projektů.

Následujícím krokem v rozvoji vyvinutého frameworku bude rozšíření zabudovaného administračního rozhraní ve spolupráci s firemním specialistou na použitelnost. Další možnosti rošíření jsou zejména ve zvýšení integrace vrstvy pro persistenci dat a vytváření prototypových modulů řešících nejběžnější požadavky klientů.

Při řešení jsem především získal široký přehled o postupech a technologiích používaných při výrobě webových aplikací. Tyto nově nabyté znalosti jsem využil v praxi již během doby psaní této diplomové práce.

10 Literatura

[1] SHERIFF, Paul D. *Introduction to ASP.NET and Web Forms* [online]. 2001-01, [cit. 2010-05-04]. <http://msdn.microsoft.com/en-us/library/ms973868.aspx>

[2] *HttpHandlers - MSDN, .NET Framework Developer's Guide* [online]. c2010, [cit. 2010-05-04]. <http://msdn.microsoft.com/en-us/library/5c67a8bd(v=VS.71).aspx>

[3] *Localizing ASP.NET Web Pages By Using Resources - MSDN, .NET Framework 4 - ASP.NET* [online]. c2010, [cit. 2010-05-04]. <http://msdn.microsoft.com/en-us/library/ms228208.aspx>

[4] *XLIFF 1.2 Specification* [online]. 2008-02-01, [cit. 2010-05-04]. <http://docs.oasis-open.org/xliff/xliff-core/xliff-core.html>

[5] 2005-11-19. *REYNOLDS, Peter - JEWTUSHENKO Tony* [online]. [cit. 2010-05-04]. <http://xml.sys-con.com/node/121957?page=0,1>

[6] POTENCIER, Fabien. *Day 19: Internationalization and Localization - Symfony Framework* [online]. [cit. 2010-05-04]. <http://www.symfony-project.org/jobeet/1_4/Doctrine/en/19>

[7] *ISO 639-2 Language Code List - Codes for the representation of names of languages (Library of Congress)* [online]. 2008-11-07, [cit. 2010-05-04]. <http://www.loc.gov/standards/iso639-2/php/code_list.php>

[8] *The Forms Working Group* [online]. 2010-03-25, [cit. 2010-05-04]. <http://www.w3.org/MarkUp/Forms/>

[9] *Inversion of control - Wikipedia* [online]. 2009-04-29, [cit. 2010-05-04]. <http://en.wikipedia.org/wiki/Inversion_of_Control>

[10] *Entity - Wikipedia* [online]. 2010-04-24, [cit. 2010-05-04]. <http://en.wikipedia.org/wiki/Entity>

[11] *Universally Unique Identifier - Wikipedia* [online]. 2010-04-28, [cit. 2010-05-04]. <http://en.wikipedia.org/wiki/Universally_Unique_Identifier>

[12] *Cache - Wikipedia* [online]. 2010-04-28, [cit. 2010-05-04]. <http://en.wikipedia.org/wiki/Cache>

[13] DAYAN, Pini. *Convert objects to JSON in C# using JavaScriptSerializer* [online]. 2009-03-12, [cit. 2010-05-04]. <http://blogs.microsoft.co.il/blogs/pini_dayan/archive/2009/03/12/convert-objects-to-json-in-c-using-javascriptserializer.aspx>

A Případová studie 1, statický web

Případové studie jsou vytvořeny formou tutoriálu, který provede čtenáře praktickým použitím frameworku, a cílem je nabídnout rychlá základní řešení, která lze rozšiřovat metodou pokus-omyl. Proto jsou psány tak, aby k jejich použití a pochopení nebyla třeba znalost struktury frameworku z předchozích kapitol, ani rozsáhlejšího teoretického podkladu vývoje a výroby webových aplikací.

Předpokládá se alespoň základní znalost ASP.NET WebForms (tedy klasického stylu vytváření stránek v ASP.NET) a znalost klíčových principů vrstvené architektury ve vývoji webových aplikací. Pro rychlé seznámení je vhodná architektura Model-View-Controller popsaná stručně například na Wikipedii ([[http://en.wikipedia.org/wiki/Model–view–controller]]).

A.1 Problém téměř statického webu

Ačkoliv se to může zdát zbytečné, tak se v praxi ukazuje, že jednou z nejčastěji potřebných vlastností frameworku pro webové aplikace je možnost škálovat jej bez zásadnějších problému dolů až na úroveň téměř statických stránek. Tato vlastnost bývá také jedna z nejčastěji opomíjených.

Základním problémem je zde právě fakt, že cílem není úplně statický web. Ten by bylo snadné napsat v čistém HTML bez nutnosti se zabývat jakýmkoliv frameworkem. V těchto případech požadujeme alespoň minimální funkcionalitu, tedy nejběžněji mít layout stránky (například se společnou nabídkou odkazů, logem, ...) ve zvlášním souboru, aby nebylo potřeba při každé změně v této části opravovat všechny stránky. Toto si můžeme velice snadno ukázat na příkladu ze základního ASP.NET. Po menších úpravách a smazání C# souborů nám zůstane následující kód:

```
<%@ Master Language="C#" %>
<!DOCTYPE html PUBLIC "\textit{-}W3CDTD XHTML 1.0 Transitional//EN" "http://www.w3.org/
    TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head runat="server">
        <title></title>
    </head>
    <body>
        <div class="menu">
            <a href="/">Úvod</a>
            <a href="/kontakt">Kontakt</a>
        </div>
        <div>
            <asp:ContentPlaceHolder id="Content" runat="server">
            </asp:ContentPlaceHolder>
        </div>
    </body>
</html>
```

```
<%@ Page Title="O firmě" Language="C#" MasterPageFile="~/MasterPage.master" %>
<asp:Content ContentPlaceHolderID="Content" Runat="Server">
```

```
    <p>Naše firma byla založena...</p>
</asp:Content>
```

Nyní ovšem přichází nejčastější nepříjemnost, tj. potřeba rozšíření vyrobeného projektu o dynamické prvky. Tyto změny pak mohou vést k zásadním komplikacím, někdy dokonce i k potřebě předelat celý projekt na jiný framework, který bude schopen poskytnout potřebné vlastnosti. V praxi se potkáváme nejčastěji s následujícími:

1. Search Engine Optimization - skládá se především ze dvou důležitých částí: potřeby změnit jednotlivým stránkám URL a propagovat doplňkový obsah jako třeba text nadpisu <h1> nebo drobečkové navigace z konkrétního vnitřku stránky (v tomto příkladu soubor .aspx) do obecného obsahu layoutu (soubor .master).

2. Požadavky typu „všechny stránky budou mít nabídku takto, akorát tady bychom to potřebovali trochu jinak“.

3. Doplnění systému pro správu obsahu (CMS), většinou pro poměrně malou část celého obsahu, například možnost přidávat fotografie do původně statické fotogalerie.

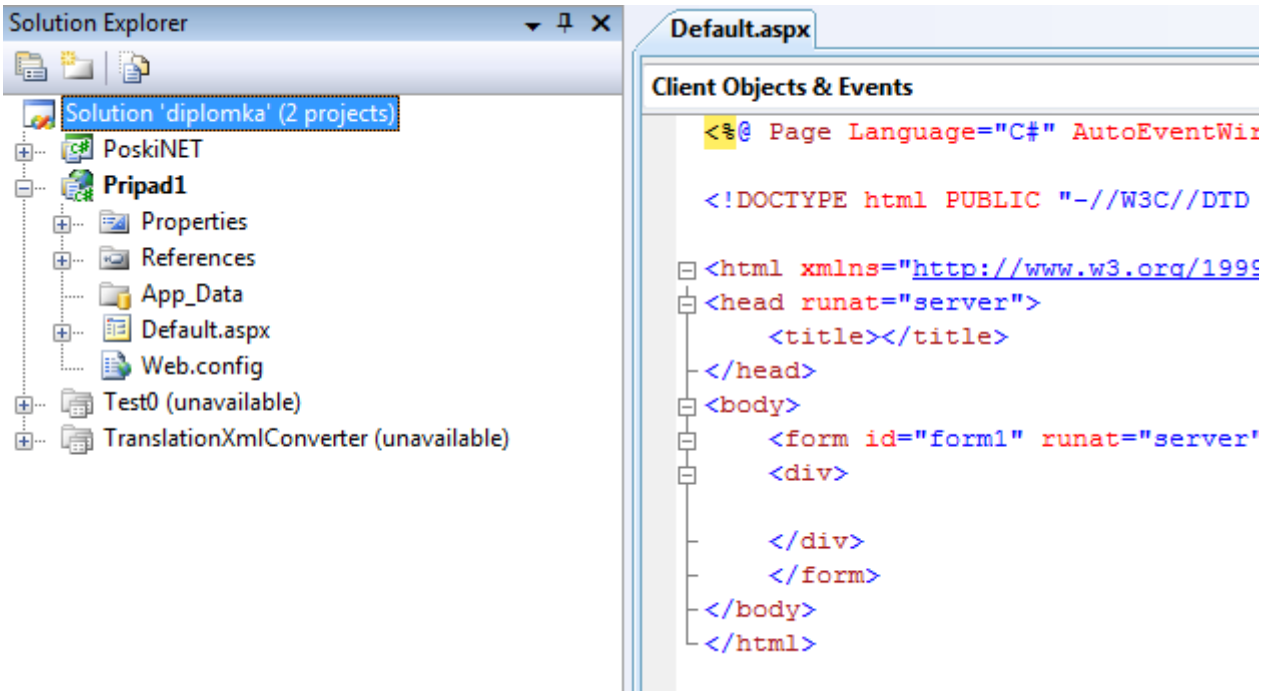
A.2 Vytvoření kostry

Nejčastějším způsobem jak začít vytvářet projekt, je použít předpřipravenou kostru. V tomto případě se tomu ovšem vyhneme a vytvoříme tuto kostru ručně z několika menších kousků kódu. Vytvoříme tedy prázdný projekt typu ASP.NET Web Application, čímž získáme strukturu jako v následující obrazovce.

Začneme připojením PoskiNET frameworku, tedy přidáme k projektu referenci na příslušné assembly a následně upravíme konfigurační soubor Web.config tak, aby při požadavku předal kontrolu toku do frameworku. K tomu nám stačí nahradit obsah souboru následujícím XML:

```
<?xml version="1.0"?>
<configuration>
    <system.web>
        <compilation debug="true"/>
        <httpHandlers>
            <add verb="*" path="*" validate="false"  type="PoskiNET.InvocationHttpHandler,␣
                PoskiNET"/>
        </httpHandlers>
    </system.web>
</configuration>
```

Druhým souborem, který se ve výchozím webovém projektu nachází je Default.aspx. Tento soubor může být při určitém nastavení Microsoft IIS smazán a plně nahrazen routováním frameworku, ale protože se jedná o výchozí soubor (analogicky k index.* u webového serveru Apache), je vhodnější i toto místo ošetřit. o lze provést smazáním C# kódu a nahrazením Default.aspx následujícím kódem:



Obrázek 23: Přehled souborů v prázdném projektu

```
<%@ Page Language="C#" AutoEventWireup="true" Inherits="Test0._Default" %>
<script runat="server">
    public void Page_Load(object sender, System.EventArgs e)
    {
        PoskiNET.InvocationHttpHandler.HandleDefaultAspx(this);
    }
</script>
```

Tím jsou vyřešeny existující soubory a můžeme přistoupit k doplnění inicializace webu. Vytvoříme tedy výchozí soubor Global.asax a nahradíme jeho obsah vlastní prázdnou třídou odvozenou od PoskiNET.InvocationHttpApplication.

```
namespace Pripad1
{
    public class Global : PoskiNET.InvocationHttpApplication
    {
    }
}
```

Jak je vidět, naše vlastní inicializace není k základnímu fungování potřeba, stačí vyměnit třídu HttpApplication. Takto připravený web používá určité výchozí nastavení, které můžeme v případě komplexnějších projektů změnit, ale v souladu se základním paradigmatem frameworku se snažíme minimalizovat množství zásahů programátora. Jsme tedy nyní připraveni k vytvoření první statické stránky.

Nastavení jazyků webu, jehož vliv potkáme v následující sekci, je řešeno ve výchozí implementaci metody InitializeApplicationLocales() v rámci třídy InvocationHttpApplication, které lze vidět v následující bloku kódu.

```
    protected virtual void InitializeApplicationLocales ()
    {
        Locales.Add(new Locale("cs", ""));
        // Tento kód je výchozí implementace a není ji třeba doplňovat do kostry projektu.
    }
```

A.3 První statická stránka

A.3.1 Routování

Nejprve je třeba definovat spojení mezi URL požadavku a konkrétními soubory aplikace. Toto se provádí skrz tzv. routování, které je dnes běžným řešením tohoto problému. Pro podrobnější informace k tomuto tématu můžete přečíst příslušnou kapitolu této práce, nebo například popis tradičnějšího přístupu k routování, které je obsaženo ve frameworku ASP.NET MVC ([[http://msdn.microsoft.com/en-us/library/cc668201.aspx]]), nebo český popis pro framework Nette ([[http://doc.nettephp.com/cs/routovani]]). Pro větší srozumitelnost textu si jen připomeňme, že původní anglický termín routing je odvozen od slova route, tedy cesta.

Ve výchozím nastavení čte framework definice cest ze souboru App_Data/Router.xml, který vytvoříme s následujícím obsahem:

```
<?xml version="1.0" encoding="utf-8" ?>
<Router>
    <SimpleRoute Id="404" Pattern="404" Locale="cs">
        <RouteDirectiveTemplate Source="Pages/404" />
    </SimpleRoute>
    <SimpleRoute Id="Index" Pattern="" TemplateSource="Pages/{locale}/Index" />
    <SimpleRoute Pattern="test" TemplateSource="Pages/Test" />
    <!--
    <SimpleRoute Pattern="kocka" Locale="cs" TemplateSource="Pages/{locale}/Kocka" />
    <SimpleRoute Pattern="cat" Locale="en" TemplateSource="Pages/{locale}/Kocka" />
    -->
    <SimpleRoute Id="OFirme" Pattern="o-firme" TemplateSource="Pages/{locale}/OFirme" />
</Router>
```

Nejprve je třeba zmínit, že ačkoliv se může zdát, že jde serializaci do XML, není tomu tak. Tato syntaxe je zvolena pro snadnější přechod mezi definicí cest v XML souboru a v C# kódu, ale kvůli vnitřnímu zabezpečení cest proti neočekávaným výsledkům je tento soubor zpracovávan speciálním parserem. Více o možnostech rozšíření je v kapitole o routování.

Jak je zde vidět, použijeme pouze jednoduché cesty (třída SimpleRoute), které jsou schopny dynamicky rozpoznat a použít pouze jazyk požadavku. Vždy je vhodné zvážit jak komplikované srovnávání potřebujeme, a pokud to lze, vyhnout se vyšší úrovni cest cestám, například cestám s regulárními výrazy.

Shora vidíme nejdříve povinnou cestu ke chybové stránce 404, která se používá pokud neodpovídá požadavku žádná jiná cesta. Tady je použita rozšířená varianta s definováním direktivy RouteDirectiveTemplate. Direktivy představují sadu operací, které se provedou, pokud příslušná cesta odpovídá požadavku. Tato direktiva nastavuje šablonu, která se použije pro vytvoření odpovědi na požadavek (typicky, ne však nutně, HTML kód). Možnou alternativou je například direktiva přesměrování.

V další cestě vidíme, že je vzor pro srovávání s URL požadavku (XML atribut Pattern) prázdný, jde tedy o úvodní stránku webu s URL například http://www.example.com/. Dále si lze všimnout, že tato cesta nepoužívá explicitně direktivu RouteDirectiveTemplate, ale zápis pomocí atributu TemplateSource. Ten má sice omezenější možnosti než použití direktiv, ale pro větší přehlednost je vhodnější, neboť soubor Router.xml by se měl používat především pro statické stránky. Jak bylo již zmíněno výše, SimpleRoute umožňuje rozpoznat jazyk požadavku, který se předává jako ISO kód jazyka na začátku URL. Tato cesta by se tedy také uplatnila u požadavku na http://www.example.com/en/, ale pouze za předpokladu, že by byla angličtina doplněna následující způsobem do inicializace aplikace.

```
protected virtual void InitializeApplicationLocales ()
{
    Locales.Add(new Locale("cs", ""));
    Locales.Add(new Locale("en", "en/"));
    // pouze doplňkový kód, není v tomto bodě součástí tutoriálu
}
```

Je třeba mezi parametry konstrutoru třídy Locale rozlišovat. První označuje jazyk podle dvoupísmenné normy ISO a jím je následně při vyhodnocení nahrazen parametr locale v TemplateSource (a další činnosti). Druhý je pak prefix URL a používá se výhradně při routování. Lze tedy například definici jazyků prohodit a udělat tím z angličtiny výchozí jazyk webu.

Třetí cesta nemá explicitně uveden identifikátor, bez kterého ovšem nemůže být definována. Je jí proto přidělen a na takto definovanou cestu není vhodné se odkazovat. Tato cesta je zároveň zvláštní tím, že platí pro všechny jazyky, ale v vždy použije stejnou šablonu. Tento postup je z praktického hlediska výhodnější, jak se ukáže později při použití pokročilejších šablon.

Cesty čtyři a pět jsou uvedeny pro celkový přehled, ale momentálně máme v aplikaci definovaný pouze jeden jazyk, výchozí češtinu, takže definice cesty číslo pět by vyvolala výjimku. Nakonec je ještě přídána další cesta, kterou využijeme později.

Tím jsou připraveny základní cesty a můžeme pokročit k přípravě šablon.

A.3.2 Šablona

Abychom měli stránku kam umisťovat, je třeba vytvořit výchozí adresáře, ve kterých framework hledá šablony. Je tedy třeba vytvořit v projektu strom adresářovou cestu Templates/Pages/cs/. Pro doplnění je vhodné si připravit i výchozí adresáře pro kaskádový styl a javascripty: _css/ a _js/. Díky podtržítku na začátku názvu těchto adresářů se řadí ve výpisu nahoře, tedy mimo soubory, které framework přímo využívá. Všechny soubory jejichž cesta začíná podtržítkem jsou navíc servírovány přímo z disku, bez zpracování.

Vytváření šablon si pro větší přehlednost můžeme rozčlenit do tří částí - přídavné soubory, společné šablony a šablony konkrétních stránek.

Přídavnými soubory se v tomto kontextu myslí kaskádové styly, javascripty, obrázky, videa a další. Jak je uvedeno výše, tyto stačí umístit do vhodných adresářů a až na výjimky se jimi framework dále nezabývá. Jednou z těchto výjimek je soubor kaskádového stylu _css/main.css, který je automaticky detekován a vkládán do všech stránek mimo administraci. Je tedy vhodné jej nyní vytvořit, třeba i prázdný.

Společné šablony, které je třeba vytvořit pro každý web zvlášť ručně jsou Pages/404 a Layout. Pages/404 se chová stejně jako jakákoliv jiná statická stránka, jenom je vyžadována pro korektní funkčnost routování. Layout je stránka, která je analogická k master stránkám z ASP.NET WebForms. Je zde ovšem mírný rozdíl v tom, že framework umožňuje vkládat jenom jeden vnitřní obsah - šablonu s identifikátorem Main - a to náhradou za text <PoskiNET:MainTemplate />. Pro začátek si tedy můžeme vyrobit základní Layout.html s následujícím obsahem:

```
<div id="lead">
    <div id="logo">
        <h1><a href="/"><span class="text">priklad1</span><span title="Návrat na úvodní
            stránku"></span></a></h1>
    </div> <!-- #logo -->

    <div class="top">
        <ul>
```

```
            <li><a href="/"><span>Home</span></a></li>
            <li><a href="/o-firme"><span>O firmě</span></a></li>
        </ul>
        <div class="clearing"></div>
    </div>

    <div class="mainCol">
        <PoskiNET:MainTemplate />
        <div class="clearing"></div>
    </div>

    <div class="rightCol">
        <h3 class="grey">Rychlý kontakt</h3>
    </div>

    <div class="clearing"></div>
    <div class="footer">
        <a href="http://www.poski.com/" onclick="return !window.open(this.href)">Webdesign Poski.
            com</a> 2010
    </div>
</div> <!-- #lead -->
```

A konečně šablona úvodní stránky, kterou si můžeme naplnit libovolným HTML kódem, například

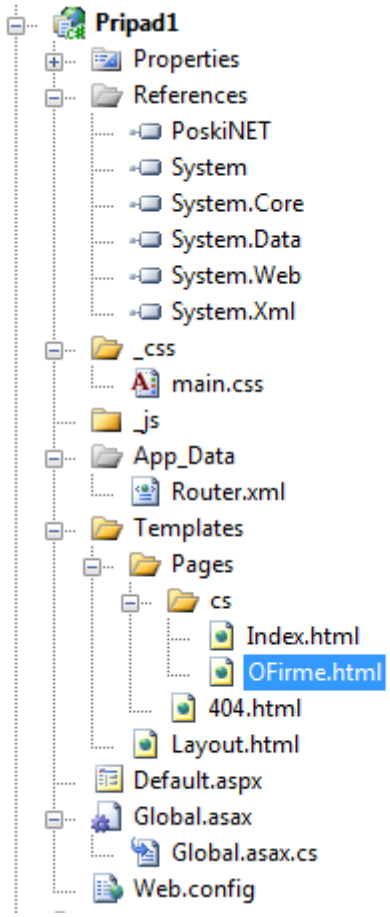
```
<h2>Případ 1</h2>
<p>
    Lorem ipsum dolor sit amet, consectetur adipisici elit , <a href="">sed eiusmod</a> ut labore
        et dolore magna aliqua.
    Ut enim ad minim veniam, exercitation ullamco laboris nisi ut aliquid ex ea commodi
        consequat
    onsectetur <a href="">adipisici elit </a> ad minim veniam. Lorem ipsum dolor sit amet,
        consectetur adipisici elit , sed emod
    ut labore et dolore magna aliqua. Ut enim ad minim veniam, exercitation.
</p>
```

Tím jsme dokončili základní funkční web se statickou stránkou. Pro rekapitulaci ještě výsledný strom projektu.

A.4 Titulky stránek

Takto vytvořený web je sice funkční, ale již na první pohled mu chybí cosi důležitého, přesněji titulky stránek. Ty jsou ve frameworku tvořeny ze tří částí - titulku stránky, oddělovače a titulku webu. Všechny tyto řetězce, včetně formátu a pořadí lze změnit, ale dnes je nejčastěji používaný přibližně tento formát „O firmě - Případ 1“ a framework jej používá jako výchozí.

Pro nastavování titulků statických stránek se využívají překladové soubory, začneme proto nakopírováním základního stromu adresáře Translations/. Tento krok je nepovinný a lze vynechat, ale později můžeme postrádat některé texty, které jsou již přeloženy. Následně doplníme obsah souboru popisujícího metainformace stránek, pro český jazyk se jedná o soubor Translations/cs/Meta.xml.



Obrázek 24: Základní strom projektu s vytvořenými šablonami

```
<?xml version="1.0" encoding="utf-8" ?>
<namespaces>
    <namespace key="/Meta/">
        <namespace key="Index/">
            <message key="Title" value="Úvodní stránka" />
            <message key="Url" value="" />
            <message key="Keywords" value="klíčové, slovo" />
            <message key="Description" value="toto je popis úvodní stránky" />
        </namespace>
        <namespace key="OFirme/">
            <message key="Title" value="O firmě" />
            <message key="Url" value="o-firme" />
            <message key="Breadcrumb0" value="" />
            <message key="Breadcrumb1" value="Index" />
            <message key="Breadcrumb2Text" value="Seznam.cz" />
            <message key="Breadcrumb2Url" value="http://odkazy.seznam.cz/Pocitace-a-
                internet/Software/" />
        </namespace>
    </namespace>
</namespaces>
```

Obsah tohoto souboru je důležitý a proto jej projdeme podrobněji. Strukturou se jedná o překladový soubor, která je popsána v kapitole o lokalizaci. Pro účely tohoto případu si jen stačí uvědomit, že jsou zde definovány metainformace pro dvě cesty (viz. routování výše v této kapitole), které jsme nadefinovali dříve.

Z hlediska SEO je nejdůležitější řádek, který definuje titulek stránky. Klíčovými slovy a popisem stránky se dnes spíše než vyhledávací stroje řídí programy pro zrakově postižené.

B Případová studie 2, netradiční výstup

B.1 Úvod do problému

Jeden z našich významnějších klientu na webu (postaveném na firemním PHP frameworku) vystavuje katalog svých přibližně tří set výrobků včetně velkého množství fotografií. Klient má k dispozici pouze propagační fotografie určené k tisku, tedy takové, které svými rozměry přesahují únosnou mez pro použití na webu a také velikost souborů nebyla zanedbatelná.

Důsledkem toho bylo, že správa těchto fotografií přes webové administrační rozhraní byla nesnadná a zdlouhavá, neboť bylo potřeba fotografii před nahráním na web nejdříve ručně zmenšit. V opačném případě se web pokusil fotografii zmenšit sám a nahrávání zhavarovalo na limitu dostupné paměti (bezpečnostní opatření v rámci PHP). Po nahrání zmenšené verze byl už web schopen vytvořit si další potřebné zmenšeniny pro náhledy sám.

Připravili jsme proto speciální aplikaci, která v několika jednoduchých krocích umožňuje zvolit z roletky produkt na webu a z disku tiskovou fotografii a sama ji pak nahraje na web v přijatelné velikosti. Zmenšování tedy probíhá na desktopovém počítači s dostatkem paměti a náš klient není zatěžován zdlouhavou a nudnou manipulací s obrázky.

Nahrávací aplikace byla vytvořena v C# a pro svižnost práce jsme použili WinForms. Brzy ovšem vyvstala otázka komunikačního formátu. Potřebovali jsme nějaký formát, který by především mohla použít i prezentace napsaná v PHP, aniž bychom museli dělat rozsáhlejší úpravy. Nakonec byl zvolen kompromis mezi cestou nejmenšího odporu a potenciálem pro budoucí rozšíření - požadavky budou posílány jako parametry v URL a odpovědi budou objekty serializované pomocí JSON. Díky tomu měla implementace serverové části jen něco přes 100 řádků.

Na tomto případu si ukážeme částečnou reimplementaci serverového řešení pomocí frameworku vytvořeného v rámci této práce. Pro odlehčení závislostí projektu bude místo specializované knihovny použita JSON serializace pomocí tříd základní knihovny dostupné v prostředí .NET [13].

B.2 Řešení

Řešení staví na základech z první případové studie a pro lepší pochopení by si ji měl čtenář nejprve projít.

Na konci tohoto případu budeme mít připravenou stránku, která bude poskytovat rozhraní pro získání seznamu aktuálních produktů ve formátu JSON.

B.2.1 Základ

Abychom nastavili stejné podmínky jako v původním problému, můžeme vyjít ze statické prezentace, která byla vytvořena v rámci první případové studie. Pokud bychom ale chtěli vytvořit nový web jen pro tento problém, potřebujeme připravit alespoň základní

minimum pro použití frameworku. Je tedy třeba přidat assembly PoskiNET do referencí a vytvořit soubory Web.config, Default.aspx a Global.asax s obsahem dle textu v první případové studii.

B.2.2 Příprava modelu

Pro tento případ nám postačuje jednoduchý model pro informační záznam o produktu. Optimálně bychom pak mohli mít tento model ve zvláštní knihovně, sdílené webovou aplikací a desktopovou aplikací na nahrávání fotografií.

```
public class MiniProdukt
{
    public int Id { get; set; }
    public string Nazev { get; set; }
}
```

B.2.3 Vytvoření vyvolání

B.2.3.1 Trocha teorie Protože vyvolání nebyla zmíněna v první případové studii, je třeba si alespoň trochu představit koncept vyvolání.

Zjednodušeně vyvolání představuje analogii controlleru s jednou akcí v návrhovém vzoru MVC a pro účely tohoto případu funguje velmi podobně jako ve frameworku ASP.NET MVC. Stejně jako controller, je i vyvolání zavoláno po dokončení routování. Rozdílem v tomto frameworku ale je, že vyvolání se nemusí aktivně podílet na určení výstupu pro uživatele.

Každá třída vyvolání definuje 3+3 složky. První trojice představuje třídy, jejichž instance budou vstupem (Parameters), výstupem (Results) a mezistupňem (State) při provádění činnosti. Druhá trojice jsou implementace třístupňové činnosti vyvolání. Jsou to příprava (Prepare), ověření dostupnosti (Check) a samotné provedení (Execute).

B.2.3.2 Implementace Nejdříve vytvoříme první trojici. V době implementace rozšíření webové aplikace již existuje seznam domluvených hodnot a jejich typů na vstupu i na výstupu. Proto si nadefinujeme třídy vstupních parametrů a výstupních výsledků podle této domluvy.

```
// Poznámka: pole této třídy budou přiřazena z parametrů v HTTP požadavku
public class SeznamMiniProduktuInvocationParameters : InvocationParametersBase
{
    public string heslo;
}

public class SeznamMiniProduktuInvocationResult : InvocationResultBase
{
    public string signatura = "api.SKRYTO.cz";
    public string verze = "1.0";
    public string kod = "chyba";
    public List<MiniProdukt> seznamProduktu = new List<MiniProdukt>();
}
```

Následně vytvoříme třídu samotného vyvolání s uvedením použitích tříd pro Parameters, Result a State. Zde je vidět, že jsme použili vlastní jenom vstup a výstup, jako mezistupeň bude použita (prázdná) výchozí třída InvocationStateBase.

Do této třídy doplníme implementaci druhé trojice. Přípravu žádnou nepotřebujeme, ale abychom mohli vyvolání úspěšně vyvolat, musíme vyvolání povolit implementací metody s atributem InvocationCheck, prázdným seznamem parametrů a návratovou hodnotou typu InvocationCheckStatus tak, jak je uvedena v kódu.

Nejsložitější část je metoda s atributem InvocationExecute, která nejdříve ověří platnost hesla a pak doplní data do výsledku. K tomu jsou použity vlastnosti Parameters a Result, jejichž typy jsou dány definicí třídy.

```
public class SeznamMiniProduktuInvocation
    : Invocation<SeznamMiniProduktuInvocationParameters,
    SeznamMiniProduktuInvocationResult, InvocationStateBase>
{
    [InvocationCheck]
    public InvocationCheckStatus CheckInvocation()
    {
        return InvocationCheckStatus.Enabled;
    }

    [InvocationExecute]
    public void ExecuteInvocation()
    {
        if (Parameters.heslo != "tajne")
            return;

        // produkty jsou vymyslene a s nicim nesouvisi
        Result.seznamProduktu.Add(new MiniProdukt() { Id = 1, Nazev = "Houpaci konik" });
        Result.seznamProduktu.Add(new MiniProdukt() { Id = 2, Nazev = "Kacenka" });
        Result.seznamProduktu.Add(new MiniProdukt() { Id = 3, Nazev = "Pexeso" });
        Result.seznamProduktu.Add(new MiniProdukt() { Id = 4, Nazev = "Elektricky vlacek" });
        Result.kod = "ok";
    }
}
```

B.2.4 Vytvoření nového potomka View

B.2.4.1 Trocha teorie Máme tedy samotný výkonný kód, ale teď je potřeba výsledek zformátovat na výstup. Výchozí třídu pro výstup (TemplateView) už jsme viděli v praxi v předchozí případové studii.

Nyní potřebujeme vlastní třídu, která vezme výsledek vytvořený vyvoláním a do HTTP odpovědi jej zapíše serializovaný jako JSON. To je vcelku jednoduché jde pouze o implementování jedné metody abstraktní třídy View.

B.2.4.2 Implementace Díky knihovní třídě System.Web.Script.Serialization.JavaScriptSerializer můžeme serializovat jakýkoliv objekt do JSON. Řešení je tedy přímočaré - serializu-

jeme přímo výsledek aktuálního vyvolání, který je dostupný přes InvocationContext.Current.Invocation.ResultI.

```
public class JsonView : View
{
    public JsonView()
    {
    }

    public override void Render(Stream stream)
    {
        if (InvocationContext.Current.Invocation != null)
        {
            System.Web.Script.Serialization.JavaScriptSerializer oSerializer = new System.Web.
                Script.Serialization.JavaScriptSerializer();
            string sJson = oSerializer.Serialize(InvocationContext.Current.Invocation.ResultI);
            byte[] bJson = Encoding.UTF8.GetBytes(sJson);
            stream.Write(bJson, 0, bJson.Length);
        }
    }
}
```

B.2.5 Doplnění routování

Všechny vytvořené části nakonec spojíme do dalšího routovacího pravidla, které zapíšeme do metody InitializeApplicationRouter() v souboru Global.asax.

Vyjdeme ze zápisu routovacích pravidel v XML souboru.

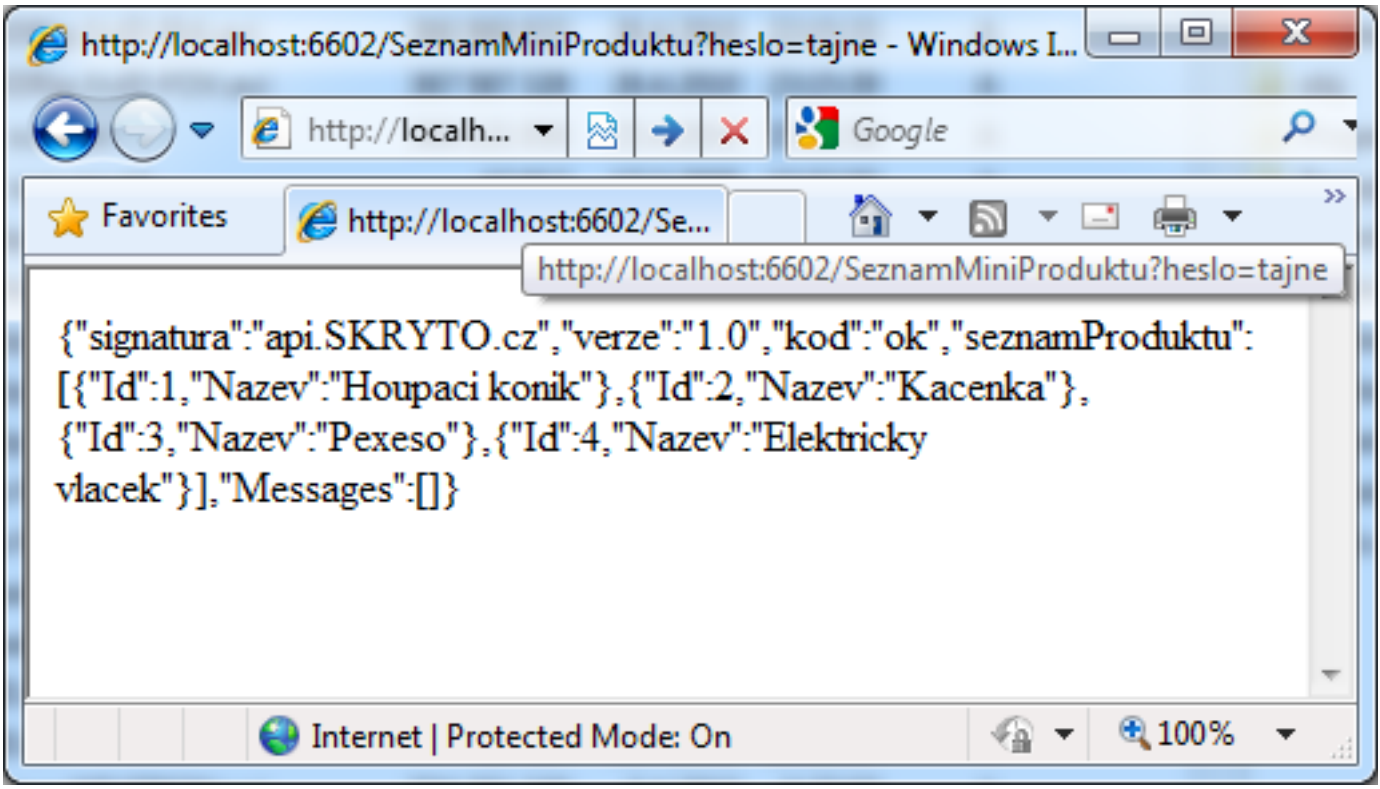
```
//new SimpleRoute(id, url)
InvocationApplication.Router.Routes.Add(new SimpleRoute("SeznamMiniProduktu", "
    SeznamMiniProduktu")
{
});
```

A tento zápis rozšíříme o nastavení dvou továren - jednu pro naše vyvolání a druhou pro náš JSON formátovač výstupu.

```
InvocationApplication.Router.Routes.Add(new SimpleRoute("SeznamMiniProduktu", "
    SeznamMiniProduktu")
{
    InvocationFactory = new InvocationFactory<SeznamMiniProduktuInvocation>()
    ViewFactory = new CallbackViewFactory(delegate() { return new JsonView(); }),
});
```

B.2.6 Výsledek

Po spuštění projektu a zobrazení URL http://localhost:6602/SeznamMiniProduktu?heslo=tajne pak uvidíme připravený výstup ve formátu JSON.



Obrázek 25: Snímek obrazovky s výsledným výstupem

C Případová studie 3, prezentace s CMS

C.1 Zadání

Cílem třetí případová studie je popsat postup znovuvytvoření firemního projektu it-poradenstvi.cz v prostředí .NET. Jelikož se jedná o prezentaci firmy zabývající se výrobou webových stránek, byl zvolen kompromis mezi vytvářením obsahu přímým zápisem HTML a použitím CMS. CMS je tedy použit pro správu novinek (a jiného častěji obměňovaného obsahu) a stránky, které jsou pro prezentaci klíčové jsou zapsány ručně a pro snadnější úpravu rozděleny po sekcích do samostatných souborů.

Prezentace k tomuto obsahuje ještě formulář pro kontaktování a přihlášení se na seminář, takže v tomto případu uvidíme i praktickou ukázku ručního vytváření šablon pro formuláře.

C.2 Kroky implementace

Pro vyřešení tohoto zadání bude třeba provést následující kroky:

1. Příprava kostry aplikace.
2. Vytvoření základních šablon a statických stránek.
3. Vytvoření formuláře a jeho šablony.
4. Příprava databáze.
5. Přidání entity a modulu uživatel.
6. Vytvoření entity novinky.
7. Vytvoření modulu novinky.
8. Vytvoření modelu a šablon pro zobrazování novinek.

Kvůli velkému rozsahu zdrojového kódu budou popsány pouze významné a zajímavé části, a proto je doporučeno studovat tento případ společně s již připraveným přiloženým kódem.

C.3 Implementace

C.3.1 Příprava kostry aplikace

Stejná jako v předchozích dvou případech.

C.3.2 Vytvoření základních šablon a statických stránek

Zde je možné použít velmi podobný postup jako v první případové studii. Je zde ovšem jedna zajímavá vlastnost šablonovacího podsystému, kterou je vhodné vyzdvihout a to post-processing. Pokud má šablona textový výstup, je po svém vykreslení ještě zpracována a může obsahovat doplňující instrukce.

Například stránka popisující e-learning je rozdělena na několik vlastních sekcí, každá z nich v samostatném souboru. Toto rozdělení ale nepřináší nic z hlediska aplikace, ale jen zpříjemňuje práci HTML kodérovi. Pokud by ovšem musel ke vkládání šablon používat C# kód, tak si spíše práci zkomplikuje.

Proto jsou doplňující instrukce zapsány pomocí méně používaného, ale validního, HTML elementu q. Následující HTML kód je jediným obsahem šablony Pages/e-learning.html. Důležité je, že všechny šablony vložené tímto způsbem procházejí opět celým cyklem vykreslení, tj. zatímco první 7 šablon jsou .html soubory, tak šablona Prihlaska je .ascx soubor, tj. s dynamickým obsahem.

```
<q class="Template">Pages/e-learning/uvodni-text</q>
<q class="Template">Pages/e-learning/vhodne-pro</q>
<q class="Template">Pages/e-learning/proc</q>
<q class="Template">Pages/e-learning/narocnost</q>
<q class="Template">Pages/e-learning/prinosy</q>
<q class="Template">Pages/e-learning/fakta</q>
<q class="Template">Pages/e-learning/kdy</q>
<q class="Template">Prihlaska</q>
```

C.3.3 Vytvoření formuláře a jeho šablony

Pro názornost budou ukázky formuláře zkráceny jen na jedno pole k vyplnění, jméno.

Jak bylo zmíněno v předchozí sekci, formulář přihlášek je řešen souborem Templates/Prihlaska.ascx. Do něj doplníme kód podle ukázek v kapitole o formulářích:

```
<%@ Control Language="C#" AutoEventWireup="false" Inherits="PoskiNET.UserControl" %>

<%
    Form form = new Form("Prihlaska");
    form.GroupMain.Add(new FormElementText("jmeno", "Jméno objednatele")).Validators.Add(new
        FormValidatorRequired());
    form.AddSubmit();

    if (form.Validate())
    {
        __w.Write("<strong>Formulář odeslán.</strong>");
        form.ShouldRender = false;
    }

    if (form.ShouldRender)
        __w.Write(form.Render());
%>
```

Důležitá informace je pro nás identifikátor formuláře předaný v konstruktoru třídy Form. Přidáním „Form“ k tomuto identifikátoru získáme název šablony, která pokud je nalezena, tak se použije k vykreslení formuláře místo implementace v metodách.

Vytvoříme tedy soubor Templates/PrihlaskaForm.ascx, do kterého doplníme HTML kód připravený HTML kodérem na základě grafického návrhu. V tomto HTML kódu nyní nahradíme některé statické části použitím metody RenderPart() různých elementů ve formuláři podle následujícího klíče.

```
// <form action="" method="post"><div>
Data.Get<Form>("form").RenderPart(FormRenderParts.ContainerBegin)

// </form>
Data.Get<Form>("form").RenderPart(FormRenderParts.ContainerEnd)

// <label for="jmeno">
Data.Get<Form>("form").Find("jmeno").RenderPart(FormRenderParts.Label)

// <input type="text" name="jmeno" id="jmeno" />
Data.Get<Form>("form").Find("jmeno").RenderPart(FormRenderParts.ElementContent)
Data.Get<Form>("form").Find("jmeno").RenderPart(FormRenderParts.Errors)
```

C.3.4 Příprava databáze

Nejdříve je třeba do projektu přidat nový soubor typu „SQL Server Database“, ideálně do adresáře App_Data/. Tento soubor následně oznámíme Castle.ActiveRecord jako místo pro persistenci dat.

Následující konfigurace je určena pro SQL Server 2008 Express. V případě jiného databázového stroje je třeba změnit parametry connection.driver_class, dialect a nejspíše také connection.connection_string. Možná nastavení jsou dostupná v online dokumentaci Castle.ActiveRecord a NHibernate.

```
<configuration>
    <configSections>
        <section name="activerecord" type="Castle.ActiveRecord.Framework.Config.
            ActiveRecordSectionHandler, Castle.ActiveRecord"/>
    </configSections>

    <activerecord isWeb="true">
        <config>
            <add key="connection.provider" value="NHibernate.Connection.
                DriverConnectionProvider"/>
            <add key="proxyfactory.factory_class" value="NHibernate.ByteCode.Castle.
                ProxyFactoryFactory, NHibernate.ByteCode.Castle"/>
            <add key="connection.driver_class" value="NHibernate.Driver.SqlClientDriver" />
            <add key="dialect" value="NHibernate.Dialect.MsSql2008Dialect" />
            <add key="connection.connection_string" value="Data Source=.\SQLEXPRESS;
                AttachDbFilename=|DataDirectory|Pripad3.mdf;Integrated Security=True;User
                Instance=True" />
        </config>
    </activerecord>
```

```
<system.web>
    <httpModules>
        <add name="ar.sessionscope" type="Castle.ActiveRecord.Framework.
            SessionScopeWebModule, Castle.ActiveRecord" />
    </httpModules>
</system.web>
```

C.3.5 Přidání entity a modulu uživatel

Modul pro uživatele - administrátory webu je již připraven i s příslušnou entitou. Stačí vytvořit třídu UserEntity s atributem ActiveRecord a tuto třídu společně s modulem zaregistrovat v Global.asax.

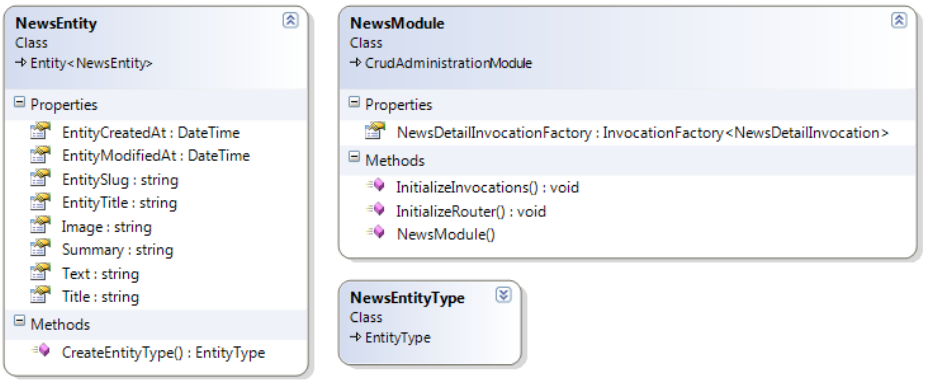
```
// Entities/UserEntity.cs
[ActiveRecord]
public class UserEntity : UserEntityBase<UserEntity>
{
}

// Global.asax.cs
public override void InitializeApplicationEntities ()
{
    Persistence.RegisterEntityType(typeof(UserEntity));
    //Persistence.CreateSchema();
}

public override void InitializeApplicationModules()
{
    InvocationApplication.Modules.RegisterModule(new UserModule());
}
```

Kdykoliv bude programátor potřebovat regenerování tabulek v databázi, stačí odkomentovat volání Persistence.CreateSchema(). Regenerovány budou pouze tabulky entitních typů, které byly do vyvolání zaregistrovány, takže vhodným dočasným přeuspořádáním příkazů lze omezit regenerování jen na některé tabulky.

Po dokončení tohoto kroku bude již možné se přihlásit do administrace, která je dostupná na adrese /admin/ relativně k URL projektu, tj. například http://localhost:6603/admin/.

C.3.6 Vytvoření entity novinky

Postup vytvoření třídy pro entitu je popsán v dokumentaci projektu Castle.ActiveRecord a na tomto místě si pouze ukážeme formou třídního diagramu vlastností, které použijeme následně v šabloně pro zobrazení.



Obrázek 26: Třídní diagram NewsEntity a NewsModule

C.3.7 Vytvoření modulu novinky

Díky tomu, že framework již obsahuje základní třídu pro modul s CRUD administrací, tak nám pro začátek stačí z této třídy odvodit vlastní třídu.

Tím jsme vyřešili administraci a vyvolání v rámci CrudAdministrationModule se postarají o základní operace s použitím výchozího nastavení. Na nás ale zbývá ještě implementace jednoho vyvolání sloužícího pro zobrazení detailu novinky.

Nejprve tedy potřebujeme vytvořit třídu vyvolání, která bude sloužit k načítání záznamu, a následně přidat továrnu na toto vyvolání do modulu.

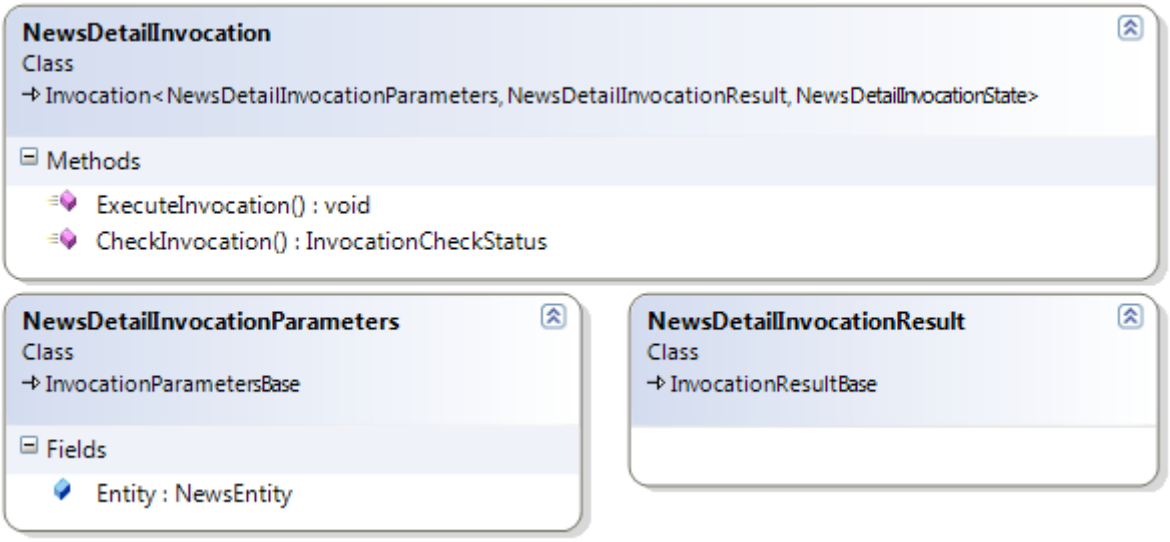
Komentovaný zdrojový kód se nachází na CD příloze v souborech Modules/NewsModule.cs a Invocations/NewsDetailInvocation.cs, šablona zobrazení detailu novinky je v souboru Templates/News/Detail.ascx.

C.3.8 Vytvoření modelu a šablon pro zobrazování novinek

Podle grafického návrhu se nám má na každé stránce zobrazovat poslední novinka s se svými atributy datum, nadpis, shrnutí a s odkazem na zobrazení detailu. Proto si navrhmene samostatnou šablonu Novinky, kterou následně vložíme na vhodné místo do šablony Layout a šablony úvodní stránky, která Layout nevyužívá.

C.3.8.1 Model V této šabloně budeme potřebovat zmíněnou poslední novinku nejdříve načíst z databáze. Abychom tuto logiku nevkládali přímo do šablony, tak si připravíme krátkou třídu modelu a nazveme ji NewsHelper. Tato třída je zajímavá z hlediska toho, jak pomocí několika málo řádků kódu lze ošetřit nejběžnější činnosti spojené s implementací webové prezentace.

```
public class NewsHelper
```

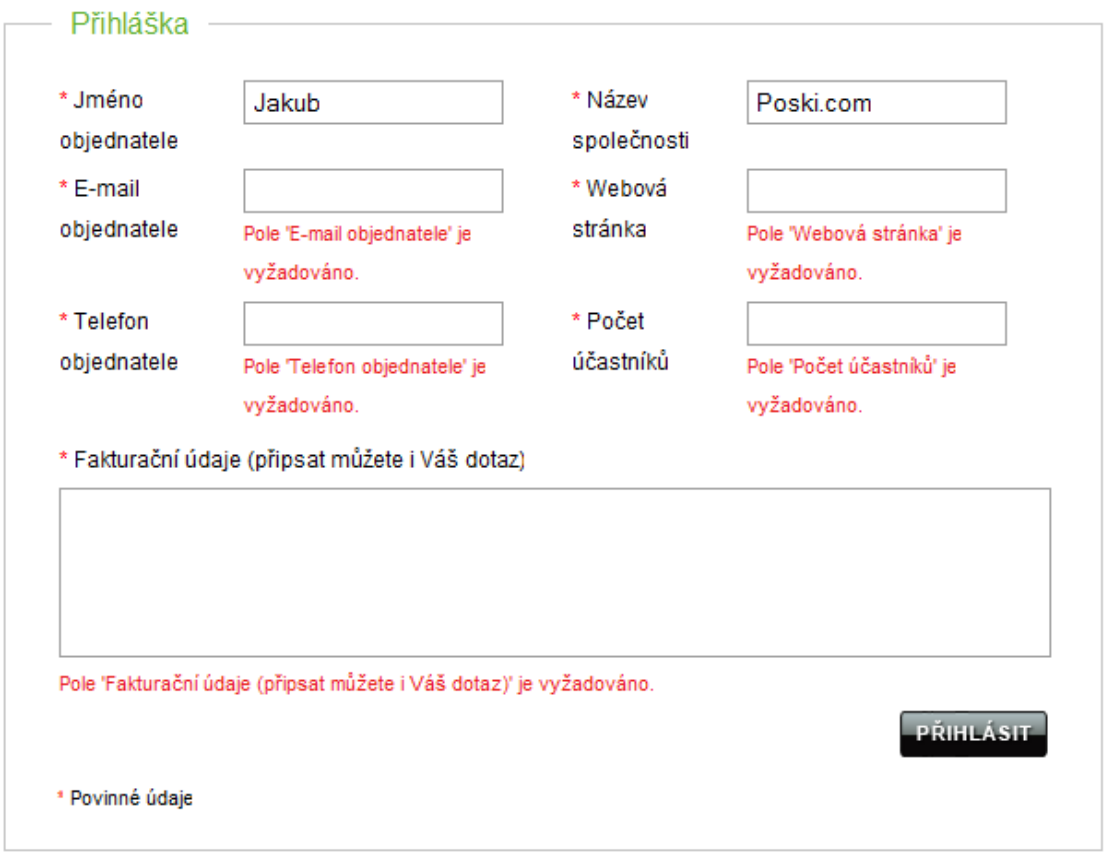


Obrázek 27: Třídní diagram vyvolání NewsDetailInvocation

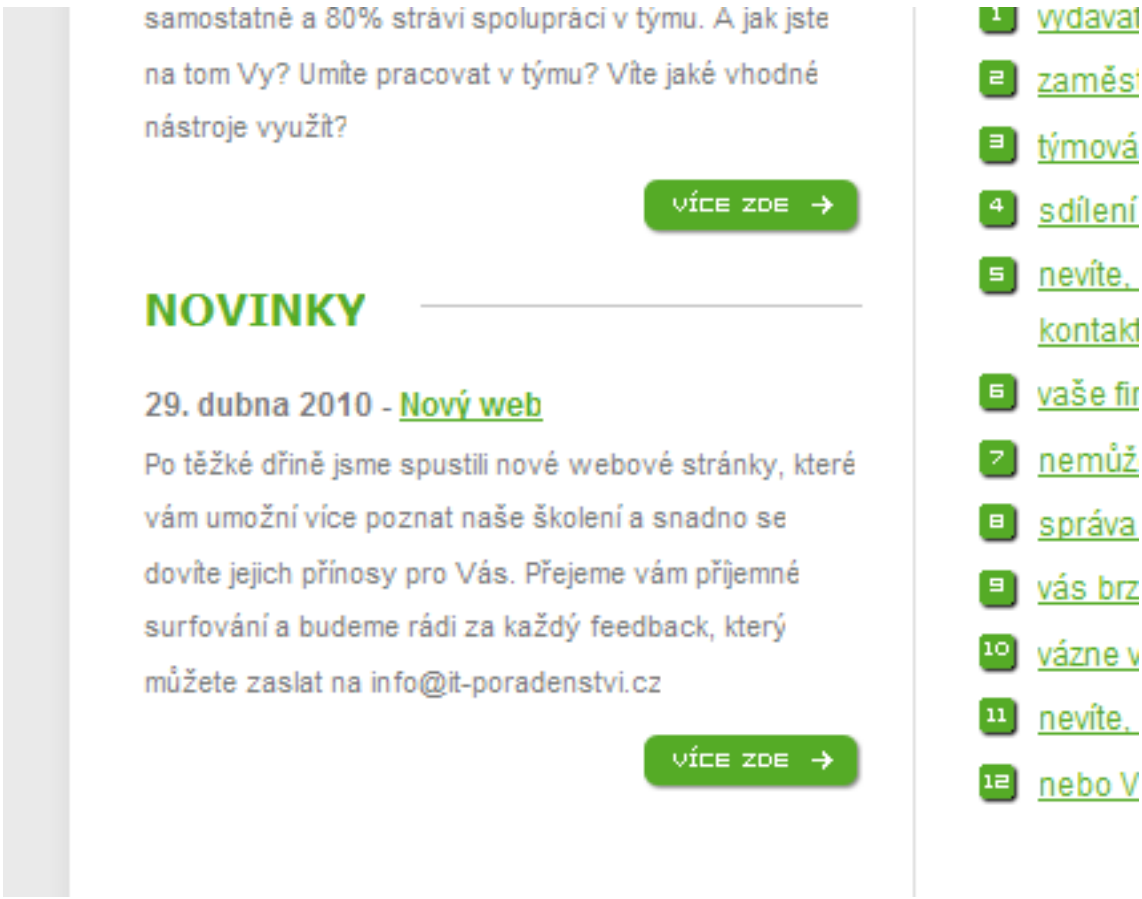
```
{
    /// <summary>
    /// Vrati pozadovany pocet novinek serazeny sestupne podle datumu vytvoreni.
    /// </summary>
    /// <param name="limit">Maximalni pocet</param>
    /// <returns>Seznam novinek</returns>
    public static IEnumerable<NewsEntity> GetRecent(int limit)
    {
        // pripravime parametr poradi
        Order[] orders = new Order[] { new Order("EntityCreatedAt", false) };
        // v novinkach najdeme hledane zaznamy
        return NewsEntity.SlicedFindAll(0, limit , orders);
    }

    /// <summary>
    /// Vrati URL pro detail novinky. URL je absolutni.
    /// </summary>
    /// <param name="entity">Novinka, pro kterou hledame URL</param>
    /// <returns>URL detailu novinky nebo prazdny retezec</returns>
    public static string GetUrlFor(NewsEntity entity)
    {
        // osetrime pripadnou nehodu tak, aby nezhavarovala cela stranka
        if ( entity == null)
            return "";
        // pripravime kolekci parametru pomoci kratsiho zapisu
        var parameters = NameObjectCollection.FromProperties(new { Entity = entity });
        // zavolame vytvoreni odkazu na routovaci pravidlo
        return InvocationApplication.Routes["News/Detail"].CreateLink(parameters);
    }
}
```

C.3.8.2 Šablona Samotnou šablonu vyrobíme jako soubor Templates/Novinky.ascx a její obsah snadno napíšeme s použitím Intellisense a modelu, který jsme právě vytvořili.

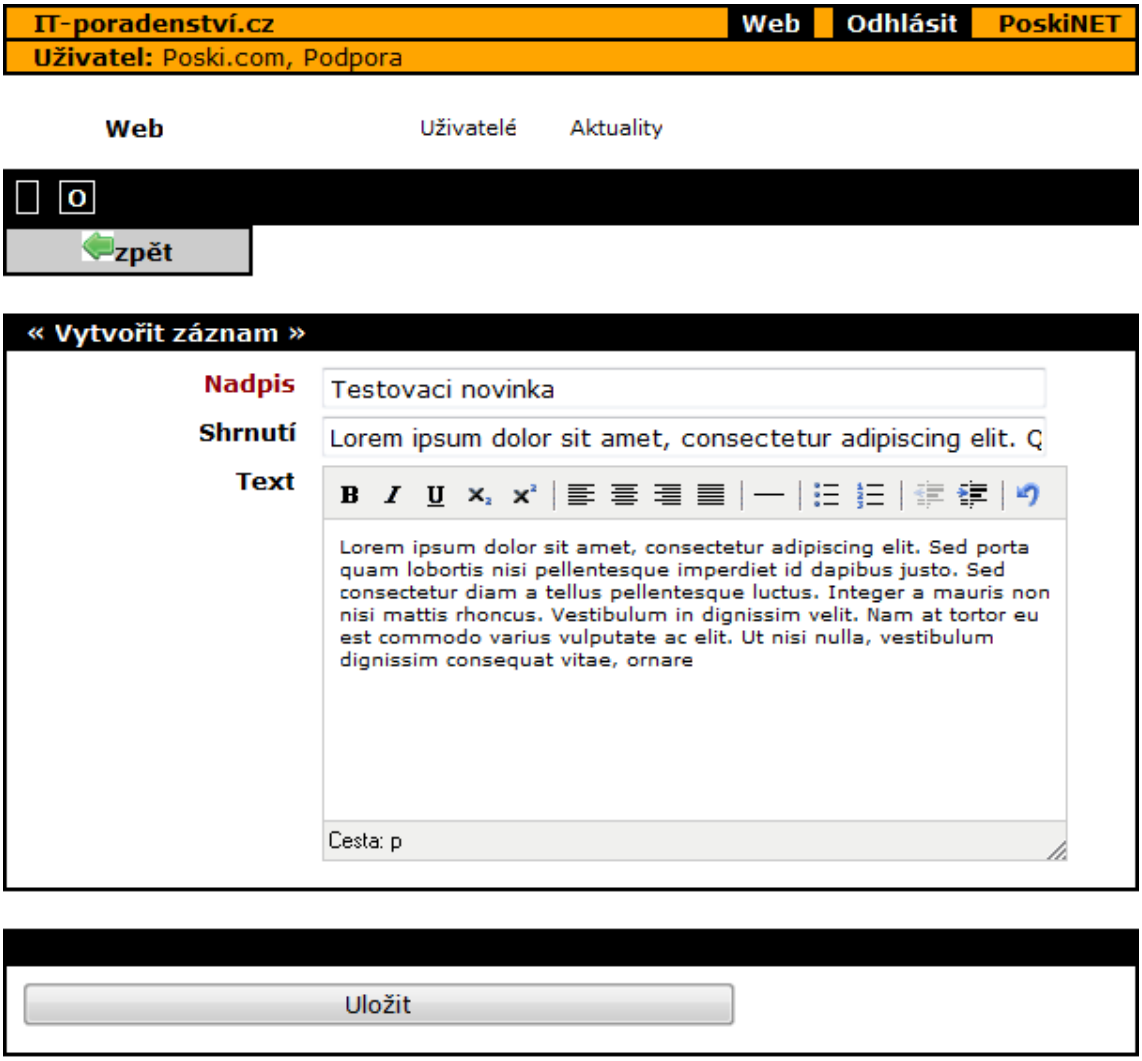
C.4 Výsledek



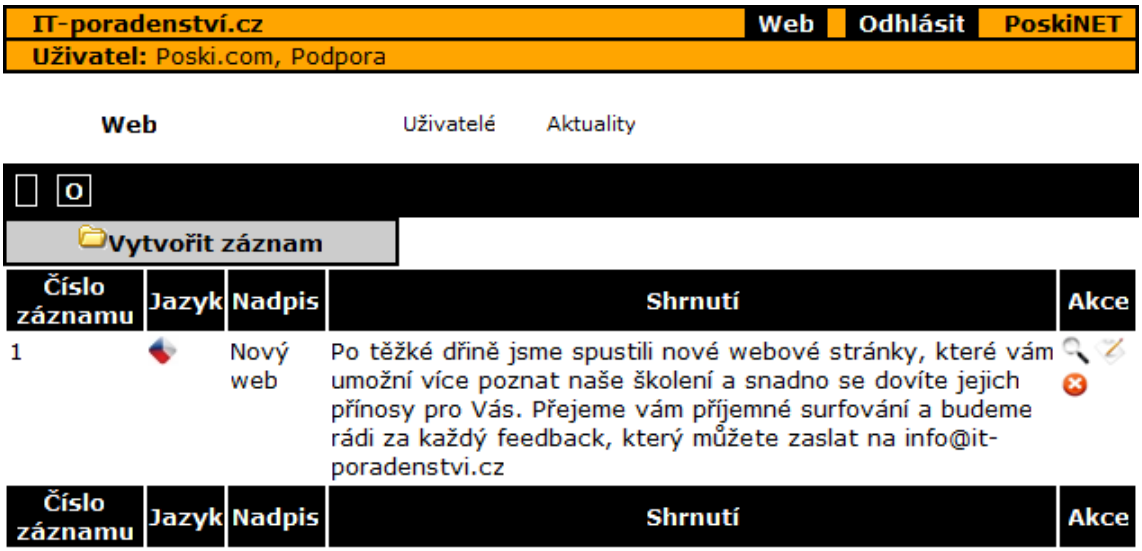
Obrázek 28: Formulář Prihlaska s ručně vytvořenou šablonou



Obrázek 29: Zobrazení poslední novinky pomocí modelu a šablony



Obrázek 30: Vytvoření nového záznamu v administraci



Obrázek 31: Výpis novinek v administraci